# Combining Heuristics for Optimizing and Scaling the Placement of IoT Applications in the Fog

Ye Xia

Communauté
UNIVERSITÉ Grenoble Alpes

# THÈSE

Pour obtenir le grade de

**DOCTEUR DE LA COMMUNAUTE UNIVERSITE GRENOBLE ALPES**

Spécialité : **Informatique**

Présentée par

**Ye XIA**

Thèse dirigée par **M. Frédéric DESPREZ**, et
codirigée par **M. Thierry COUPAYE** et **M. Xavier ETCHEVERS**

préparée au sein d'**Orange Labs** et de l'**INRIA**
dans **l'École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique**

# Combining Heuristics for Optimizing and Scaling the Placement of IoT Applications in the Fog

Thèse soutenue publiquement le 17/12/2018,
devant le jury composé de :

**M. Stéphane Genaud**
Professeur à Université de Strasbourg, Président
**M. Guillaume Pierre**
Professeur à Université de Rennes 1, Rapporteur
**M. Pierre Sens**
Professeur à Sorbonne Université, Rapporteur
**M. Adrien Lebre**
Professeur à IMT Atlantique, Examinateur
**M. Loïc Letondeur**
Ingénieur de Recherche chez Orange Labs, Examinateur
**M. Frédéric Desprez**
Directeur de recherche à l'INRIA, Directeur de thèse
**M. Thierry Coupaye**
Directeur de recherche chez Orange Labs, Co-Directeur de thèse
**M. Xavier Etchevers**
Ingénieur de Recherche chez Orange Labs, Co-Directeur de thèse

# Abstract

As fog computing brings processing and storage resources to the edge of the network, there is an increasing need of automated placement (i.e., host selection) to deploy distributed applications. Such a placement must conform to applications' resource requirements in a heterogeneous and dynamic fog infrastructure, and deal with the complexity brought by Internet of Things (IoT) applications tied to sensors / actuators. This thesis presents a model, an objective function, and heuristic algorithms to address the problem of placing distributed IoT applications in the fog. By combining proposed heuristics, our approach is able to deal with large scale problems, and to efficiently make placement decisions fitting the objective—optimizing placed applications' performance. The proposed approach is validated through complexity analysis and comparative simulation with varying sizes of infrastructures and applications.

# Contents

# Abbreviations

| | |
|---|---|
| ACO | Ant Colony Optimization. |
| AFNO | heuristic Anchor-based Fog Nodes Ordering. |
| BW | Bandwidth. |
| CPU | Central Processing Unit. |
| DAFNO | heuristic Dynamic Anchor-based Fog Nodes Ordering. |
| DAG | Directed Acyclic Graph. |
| DB | Data Base. |
| DC | Data Center. |
| DCO | Dynamic Components Ordering. |
| DSP | Data Stream Processing. |
| DZ | Dedicated Zone. |
| FailCap | heuristic Latency Failure Cap. |
| GA | Genetic Algorithm. |
| GB | Giga Byte. |
| GFlops | Giga Floating point operations per second. |
| ILP | Integer Linear Programming. |
| InitCO | Initial Component Ordering. |
| IoT | Internet of Things. |
| MBps | Mega Byte Per Second. |
| OS | Operating System. |
| PC | Personal Computer. |
| PoP | Points of Presence. |
| RAM | Random Access Memory. |
| SD | Standard Deviation. |
| WAL | Weighted Average Latency. |

# 1 Introduction

## Contents



Nowadays, the Internet of Things (IoT) envisions interconnecting every thing and person via the Internet. According to the estimation of Cisco [1], there will be 50 billions of end devices (or "things") connected to various networks by 2020. With such a huge amount of end devices that can sense / actuate to the physical environment, the IoT shapes the future interaction between human and the world. However, these end devices must generate a high volume of data to process, which makes it necessary to leverage resources beyond end devices.

Cloud computing [2], which concentrates processing and storage resources with data centers, is positioned as a key enabler of IoT applications (*i.e.,* applications tied to sensors / actuators for processing sensory data or actuating to the environment). Based on resource-rich data centers, the cloud provides an "infinite" resource pool, and can be used to overcome shortcomings of resource-constrained end devices. However, being located in the core network, data centers are far from end devices (in terms of network latency), which makes the cloud unsuitable for time sensitive IoT applications. Furthermore, the huge amount of sensors are likely to generate high volume of data. Transferring all sensor-generated raw data to the cloud can congest the core network.

Motivated by IoT applications that require low response times, data privacy enforcement, and the control over the amount of data commuting by the core network, fog computing [3, 4] extends the cloud by making use of devices close to sensors and actuators (*i.e.,* devices in the edge network layer and even end devices). Composed of devices distributed in different network layers, the fog makes it possible to process data locally (*i.e.,* in devices closer to the data's sensors than the cloud), which helps to lower applications' response time. Moreover, high-volume sensory data can be filtered and aggregated through local analytics. In this way, only post-analysis data needs to be sent to the cloud (for storage and further mining), which reduces the bandwidth consumption in the core network.

## 1.1 Motivation

To take the advantage of local resources provided by the fog, a proper decision of where to place applications (*i.e.,* how to select applications' hosts) must be made. Such placement decisions impact both applications' performance and the hardware resource consumption. However, known to be an NP-hard problem [5, 6], applications' placement decision-making in the context of IoT and fog exhibits the following challenges:

- **Heterogeneity**: the fog contains a large number of heterogeneous devices connected to various networks. These devices' resource capacities, network positions, and privacy / hardware / software features strongly differ.
- **Constraint diversity**: to be executed properly, IoT applications must conform to many kinds of constraints, which are related to consumable resources (*e.g.,* processing and bandwidth capacities), non-consumable properties (*e.g.,* network latency, privacy), and different entities (*e.g.,* software elements and communication channels, which compose applications).
- **Locality and geo-distribution**: to process sensory data locally, an IoT application need to be localized (*i.e.,* be placed in devices close to sensors / actuators that the application tied to). However, as sensors / actuators are spread over different geographical locations, an IoT application can span multiple localization areas, which complicates the application's localization.
- **Scalability**: to be reactive to applications' deployment requests, placement decisions must be made time-efficiently. However, the complexity of placement problem dramatically increases with the fog's and applications' sizes, which makes it hard to deal with large-scale problems.
- **Dynamicity**: because of applications' arrival / departure and volatile end devices (*e.g.,* devices' mobility / churn), the fog and applications to place have a constant varying and unpredictable nature.

The fog and IoT bring new challenges (such as aforementioned ones) for applications' placement. Placement approaches for placing applications in the cloud, which have been investigated in many works, are not designed to and can not deal with these challenges. As a result, new mechanisms must be developed to address them.

## 1.2 Contribution

This work deals with the problem of placing IoT applications in the fog, and aims at optimizing placed applications' performance. As stated in Section 1.1, the fog is highly dynamic, which makes it necessary to continuously

update a placement decision to adapt to dynamic changes. Thus, this work solves the placement problem with two phases:

- *initial placement*, for making an initial placement decision when no application is placed;
- *dynamic placement*, for dynamically adjusting the placement decision so as to adapt to dynamic changes.

To solve the placement problem and to overcome challenges discussed in Section 1.1, this work makes the following contributions:

- An initial placement decision-making mechanism. More precisely, this mechanism contains:
  - a model that formulates the problem of placing a set of IoT applications in a fog infrastructure;
  - an objective function that aims at optimizing applications' performance;
  - a naive algorithm which guarantees to find a placement that satisfies considered applications' requirements if such a placement exists;
  - five combinable heuristics that accelerate the placement decision-making process, make the placement algorithm much more scalable, and improve placement result quality according to the objective function.
- A complexity analysis and a simulation-based evaluation of the proposed initial placement mechanism. The evaluation compares different heuristic combinations and heuristics' parameter settings from two aspects: i) scalability in terms of the problem scale that an algorithm can deal with within a timeout; ii) result quality in terms of placed applications' average response time.
- A dynamic placement decision-making mechanism, which extends the initial placement decision-making mechanism to deal with the dynamicity of the placement problem. This mechanism combines: i) a placement re-optimization approach, which purposes to keep the placement close to the optimum; ii) a placement repairing approach, which rapidly repairs the placement when certain constraints are violated because of dynamic changes.
- An evaluation of the proposed dynamic placement mechanism, which compares a set of dynamic placement algorithms under different types of dynamicity (*i.e.,* applications' arrival, devices' mobility / churn).

These contributions lead to two publications [7, 8].

## 1.3 Outline

The rest of this thesis is organized as follows. Chapter 2 details this work's background. The initial placement problem introduces a lower complexity and is discussed first in Part I. In detail, Chapter 3 gives the initial placement's state of the art, which discusses *exact algorithms*, *metaheuristics*, and *heuristics* proposed in related works for solving the initial placement problem; Chapter 4 proposes our mechanism for making initial placement decisions, which gives our model, naive placement algorithm, accompanied heuristics, and an analysis of the algorithm complexity; Chapter 5 evaluates the proposed initial placement approach, which compares different combinations of heuristics proposed in Chapter 4 based on two use cases. Based on the initial placement approach proposed in Part I, Part II tackles the dynamic placement problem. In detail, Chapter 6 gives the state of the art for the dynamic placement; Chapter 7 proposes our mechanism for making dynamic placement decisions, which gives an approach that dynamically re-optimizes the placement, an approach that make placement decisions rapidly when the placement is no longer valid because of the dynamicity, and these two approaches' combination; Chapter 8 evaluates the proposed dynamic placement mechanism. Finally, Chapter 9 concludes and discusses future works.

# 2 Background

Contents



This chapter gives basic concepts of Internet of Things (IoT), fog computing, and applications' placement.

## 2.1 Internet of Things

### 2.1.1 Overview

The IoT shapes a paradigm in which the physical environment around us is embedded with sensors / actuators connected to the Internet [9]. Sensors convert physical parameters into digital data, from which we can derive to information / knowledge of the physical environment. Actuators transform logical decisions to physical actions, based on which the physical world can be affected.

As illustrated in Figure 2.1, an IoT ecosystem contains both hardware devices and software applications. IoT applications analyze sensory data gathered from sensors, and realize actions to the physical world through actuators. An IoT application can have multiple sensors and actuators cooperating to get environmental information and to realize reaction decisions made by the application. Considering that the IoT is a world-wide network of interconnected devices, devices' cooperation makes many innovative IoT applications possible. Some IoT application fields are discussed in the next subsection.

### 2.1.2 Application Fields

The IoT affects how people interact with the environment in many fields. Two representative examples—smart home and smart transportation are detailed in the following.

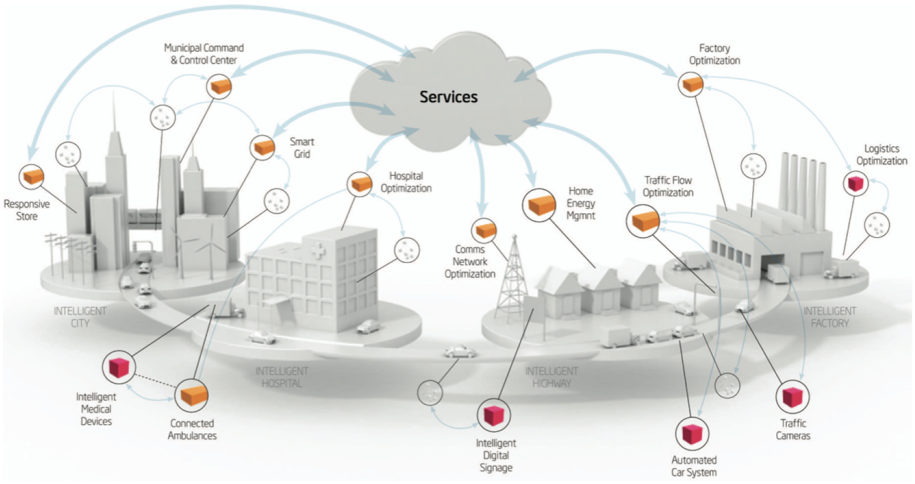


Figure 2.1: IoT Ecosystem Schema (containing software applications and a huge number of heterogeneous devices).
Source: http://siliconangle.com

### Smart Home

With household sensors, a wide range of environmental parameters (*e.g.,* temperature, humidity, air quality, door open / close, human / pet presence) can be sensed, which allows applications to better understand the home environment, and to make the physical condition of a home more comfortable [10, 11]. For example, the lighting can be automated in a various context, such as: turning off corresponding lights when a room is not occupied, or when there is enough sunlight; turning on all lights (and activate an alarm) when encountering security issues or accidents (*e.g.,* fire); adjusting window shades according to inhabitants' current states (*e.g.,* reading, taking a rest, etc). Similar services based on IoT include automated heating, patient / aging monitoring, voice-based control, and so on.

With household devices connected to the Internet, a home's information can be shared with neighbors, which brings the intelligence to the neighborhood [12] and allows providing services such as public security / emergency service, healthcare service within the community, and so on. A detailed IoT application example in this field is discussed in Section 5.2.

### Smart Transportation

By equipping vehicles and roads with sensors, the IoT paradigm helps to gather traffic information, avoid traffic jams, assist the driving, and finally realize an intelligent traffic control [10]. Moreover, supply chains can also be optimized based on the IoT [10]. By monitoring transported goods' quality, realtime distribution decisions can be made during the transport. Such decisions allow timely delivering perishable goods such as fruits and meat,

which results in less wasting. A detailed smart transportation application is discussed in Section 5.3.

## 2.2 Fog Computing

Today, there are billions of end devices connected to the Internet. However, despite the huge amount, end devices are constrained by kinds of physical constraints concerning temperature, physical space, and so on, which makes them suffer from limited processing / storage resource capacities[1]. Because of such limits, a single end device does not suit to carry out calculation-intensive data analysis. Considering that almost all sensory data is generated by end devices, these data must be transferred to (and analyzed in) devices capable enough. Being able to fulfill such a requirement of resources, two infrastructure paradigms—cloud computing and fog computing have been positioned as key enablers of IoT applications, and are introduced in the following.

### 2.2.1 Overview: from Cloud to Fog

Cloud computing [2], which enables a ubiquitous and on-demand network access to a resource pool, allows deploying applications in Data Centers (DC). Based on resource-rich DCs, a cloud has a huge amount of processing / storage resources, whose capacity can be considered as "infinite". By providing an "infinite" resource pool over the Internet, cloud computing makes up shortcomings of resource-constrained end devices in the IoT paradigm.

Although the cloud is highly capable, it has to face substantial challenges for deploying time-sensitive / bandwidth-hungry / privacy-sensitive applications:

- Being located in the Internet backbone, clouds' DCs are far from end devices in terms of network latency[2], which makes the cloud unsuitable for time-sensitive IoT applications.
- As illustrated in Figure 2.2, end devices must go through the core network to communicate with clouds. Because available bandwidth of the core network is limited, if bandwidth-hungry IoT applications keep sending high-volume data to clouds, the core network will be congested.
- The privacy of applications deployed on the cloud can be risked because: i) a public cloud, which is shared by multiple users, can not guarantee that personal data stored on the cloud is always kept private; ii) data transferred to the cloud must pass by many networks (or network links), which results in a large surface to be attacked.

[1] Some sensors / actuators even do not have any processing or storage resources.
[2] The network latency between a DC and an end device can be hundreds of milliseconds.

Motivated by these applications, another hardware paradigm—fog computing is proposed.

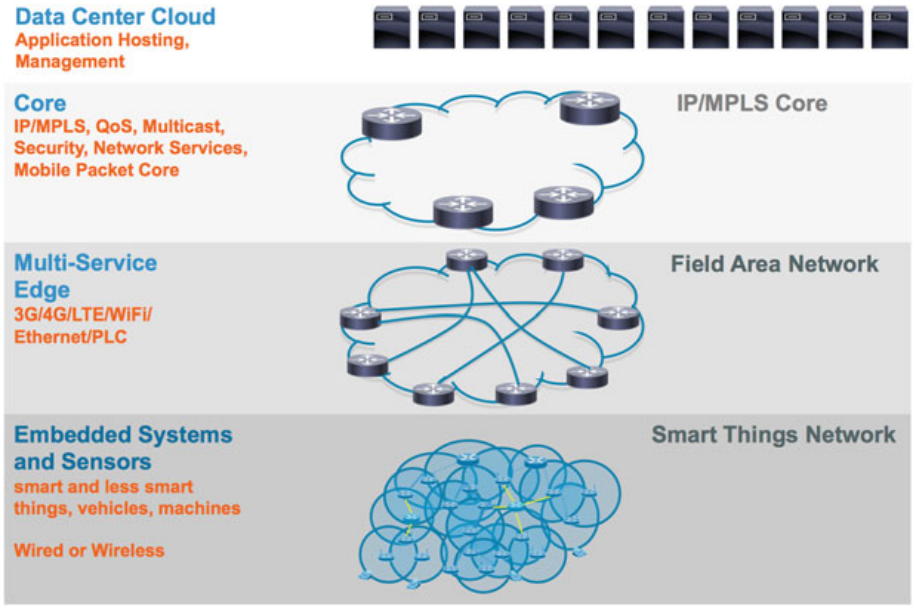


Figure 2.2: Network Positions of Cloud, Edge Devices, and End Devices. Source: [13]

In order to extend the cloud to "ground" level, fog computing [3, 4] makes use of devices in lower network layers, including devices in the edge (*e.g.,* routers, gateways) and end devices. Based on these devices, which are closer to sensors / actuators than DCs, the fog gains the following advantages:

- *low network latency.* By deploying IoT applications near to their sensors / actuators, sensory data can be analyzed locally, which helps to lower the time spent for transferring the data[3].
- *bandwidth preservation.* By analyzing / filtering / aggregating data locally, the fog can avoid transferring raw data to DCs, which helps to preserve the bandwidth of the core network.
- *reinforced privacy.* By using end devices, the fog allows storing / processing an end user's data in her / his own devices, which better protects personal data's privacy.
- *high availability.* A fog contains a huge number of geographically distributed devices. Applications deployed in the fog can be replicated in devices for serving different geographic regions. In this case, when a device is down, the other devices can provide a fast fail-over, which helps to avoid the problem of single point of failure.

[3]The network latency between a fog device and a sensor / actuator can be tens of milliseconds or even lower.

Edge / end devices help to overcome drawbacks of the cloud. However, unlike resource-rich DCs, an edge / end device can not be as capable as a DC. Moreover, caused by end devices' volatility (*i.e.,* unpredictable churn / mobility), end devices are not as stable as the cloud. Consequently, edge / end devices are rather an extension than a replacement of the cloud.

Fog computing integrates cloud and edge / end devices, and allows taking advantages of both of them. With "infinite" resources, the cloud is suitable for intensive / time-insensitive processing and high-volume / permanent storage. On the other hand, time-sensitive / bandwidth-hungry / privacy-sensitive applications can be deployed in devices in lower network layers. In this case, local analytics can be performed in edge / end devices to make real-time decisions, and provides local storage to improve applications' performance. Moreover, raw data generated by sensors can be filtered and aggregated in edge / end devices, which allows transferring only the data to be further mined to the cloud, and lowers the bandwidth consumption in the core network. Such a cooperation of devices in different network layers allows them to overcome each other's bottlenecks and mutually benefit from each other. Thus, compared with the cloud, many more applications can be satisfied by fog computing.

### 2.2.2 Related Technologies

This subsection discusses several technologies related to cloud / fog computing, as detailed in the following.

#### Distributed Application

A *distributed application* is an application that can be run on multiple networked devices [14]. Such an application is decomposed into multiple functional and deployment units, namely *software elements*, which communicate with each other via *communication channels* and carry out calculation to realize the application's monolithic functionalities. Because of the distribution nature, a distributed application can make use of resources of different devices, and suit well distributed infrastructures such as cloud and fog.

#### Virtualization

Based on the *virtualization* technology [15], a device can be logically divided into multiple Virtual Machines (VM) / containers. Each VM / container uses only a predefined (and configurable) share of this device's resources. In this way, different software elements can be deployed on VMs / containers of a same device to share its resources. Thanks to the virtualization, cloud / fog computing arrives to be an open and extensible architecture enabling third party developers to deploy their applications and to share the infrastructure's resources.

### Live Migration

Live migration [16] refers to the process of moving a running VM / container between two different devices, which relies on: i) transferring memory / disk contents of the VM / container from the source device to the destination device; ii) stopping the original VM / container, transferring memory / disk changes, and starting the cloned VM / container, which implies a temporal pause of the VM / container. Based on live migration, a VM / container can be migrated without disconnecting entities communicating with it, and deployed applications are made highly reconfigurable.

### Middleware

As depicted in Figure 2.3, to run applications on an infrastructure, a middleware is needed to link software applications with hardware devices [10]. Such a middleware is a software layer consisting of three sub-layers:

- *application composition*, which instantiates (or constructs) and models applications to deploy;
- *application management*, which manages each application's life cycle in an infrastructure (*i.e.,* deployment, reconfiguration, deletion, etc);
- *device abstraction*, which models hardware devices.

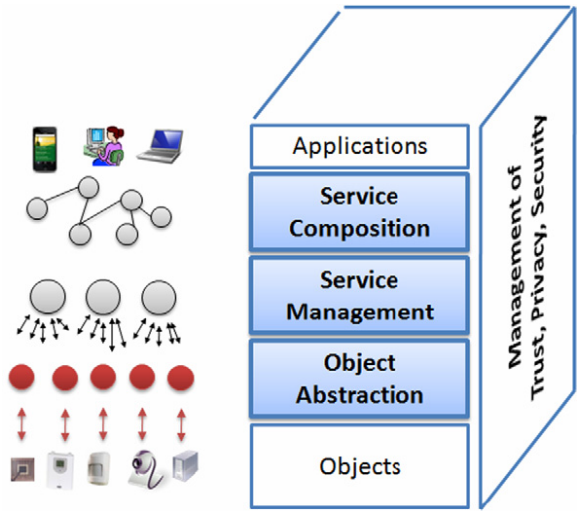


Figure 2.3: Middleware Architecture.
Source: [10]

Such a middleware is provided by and integrated with the cloud / fog, which simplifies applications' operations, and allows users of the cloud / fog to focus on their applications' core business.

## 2.3 Applications' Placement

*Applications' placement* plays a key role in "application management" of cloud / fog's middleware (see Figure 2.3), which indicates where to deploy considered applications (*i.e.,* selecting a hosting device for each software element) [17]. Considering that a component is possible to be hosted by any device, there can be many possibilities for placing a set of applications in an infrastructure. Among these possibilities, one placement must be chosen as a decision, according to which applications are deployed. The problem of how to make such placement decisions is referred to as the placement problem.

A placement decision impacts both placed applications' performance and the infrastructure's hardware resource consumption [18]. Given a placement problem, its placement decision should be (or be close to) the optimal one (regarding the problem's optimization objective, *e.g.,* optimizing applications' performance). Thus, the placement problem can be considered as an optimization problem. To ensure placed applications' proper execution, enough resources must be provided to each software element / communication channel. Thus, the placement problem is also a search problem with constraints, and a valid decision must conform to all constraints [19, 20].

A placement problem can be further complicated by devices' mobility / churn and applications' arrival / departure, which make the placement problem (*i.e.,* the infrastructure and applications to place) changes dynamically. In this case, the placement problem can be divided into the following two sub-problems [21, 22]:

- *initial placement problem*, whose placement decisions are made initially when no application is placed.
- *dynamic placement problem*, whose placement decisions are made for adjusting the current placement (*i.e.,* changing certain software elements' hosts).

The initial and dynamic placement problems can be specially dealt with, which allows them to be solved more efficiently [23, 24].

# Part I

# Initial Placement

# 3

# State of the Art of Initial Placement

Contents



This chapter gives initial placement's state of the art. Section 3.1 describes the initial placement problem, and introduces several criteria that must be considered for placing IoT applications in the fog. Section 3.2 discusses related works.

## 3.1 Problem Description and Criteria

As stated in Chapter 1 and Chapter 2, dealing with a placement problem is for making a decision of mapping a set of software elements (*i.e.,* components) onto a set of hardware devices. All possible mappings (*i.e.,* placements) compose a search space, within which one placement must be selected as the decision. In the initial placement problem, the decision is made initially when no application is placed. For designing an approach dealing with this problem in the context of fog and IoT, several criteria must be considered, as detailed in the following.

**Result Quality**. A placement problem is a search / optimization problem with an optimization objective (*e.g.,* optimizing applications' performance / minimizing the infrastructure's resource consumption). Ideally, given such a problem, the optimal placement should be returned. However, for accelerating the decision-making process, many approaches does not guarantee to return the optimum. How much an approach's results fit the optimization objective strongly impacts placed applications' performance / the infrastructure's resource consumption. For discussing different approaches' result quality, studied works are classified into four levels:

- ++ : returning the optimal result;
- + : being able to get close to the optimum;
- − : risking of trapping into a local optimum;

- $--$ : returning a result with random quality.

**Scalability**. To be reactive to applications' deployment requests, placement decisions must be made time-efficiently. However, the placement problem is proven to be NP-hard [5, 6], which makes it hard to deal with large scale problems. Under a timeout, an approach that is able to deal with larger placement problems (*i.e.,* to place more components in an infrastructure with more devices) is more scalable. Generally, an approach's scalability is strongly related to the result quality it guarantees. For discussing different approaches' scalability, studied works are classified into four levels:

- $++$ : applying certain approaches specialized for accelerating the search (without taking care of the result quality);
- $+$ : searching for a valid placement around a placement with satisfactory quality;
- $-$ : continuously refining the result with a timeout (or until the result can not be significantly improved);
- $--$ : continuing the search until the optimum is guaranteed to be found out.

**Constraint Coverage**. To ensure that placed applications can be executed properly in the fog, a valid placement must conform to several kinds of constraints:

i) each component must be provided with enough IT resources (*i.e.,* CPU, RAM, DISK).
ii) each component must be placed in a device with required properties (*e.g.,* privacy, OS, etc).
iii) each communication channel (via which components communicate with each other) must be provided with enough network resources (*i.e.,* network latency, bandwidth).

Many approaches in the context of cloud only deal with constraint i). However, for placing IoT applications in the fog, more constraints must be covered. Considering that devices in the fog are highly heterogeneous (*e.g.,* belong to different users, have different OS), constraint ii) must be taken into account for satisfying applications requiring specific properties (*e.g.,* a privacy-sensitive component must be placed in its owner's devices). Considering that the fog contains resource-constrained links, constraint iii) must be taken into account for satisfying time-sensitive / bandwidth-hungry applications.

**Location Dependency**. Based on sensing / actuating services, an IoT application is tied to its sensors / actuators, whose locations must be considered when making placement decisions.

**Heterogeneity**. Devices in the fog are of different resource capacities, located in different network layers, and connect to different networks. Such heterogeneity must be supported as well.

These criteria are used to evaluate each related work detailed in Section 3.2.

## 3.2 Related Works of Initial Placement

Among studied works, [25, 6, 24, 26, 27] focus on fog computing; [28] is generic enough to suit both fog and cloud. From applications' point of view, [25, 6, 26, 27] deal with IoT applications. As fog computing is still a recently emerging research topic, other related works are rather in the context of cloud. Nevertheless, they are still worth to discuss.

The related works are classified into exact algorithms, metaheuristics, and heuristics, and are respectively discussed in Section 3.2.1, Section 3.2.2, and Section 3.2.3. Finally, these works are summarized and compared in Section 3.2.4.

### 3.2.1 Exact Algorithms

[25] deals with the problem of placing IoT applications in an infrastructure containing cloud DCs and edge devices. Based on Integer Linear Programming (ILP), this problem is expressed with mathematical constraints and an objective function. With mathematical constraints, [25] ensures that each component gets enough CPU, RAM, and DISK resources, and that each communication channel's network latency can be accepted for properly executing considered applications. In order to make placed applications more reactive, this approach applies an objective function of maximizing the number of components placed in the edge (rather than in the cloud). A problem expressed with ILP can be solved by generic ILP solvers, such as CPLEX [29]. Given a placement problem, starting from a random initial placement, ILP solvers continuously approaches the optimum until no better placement exists (*i.e.,* the optimal placement is found), which makes [25]'s result quality classified as ++. Dealing with IoT applications in the cloud and edge, this approach supports both location dependency and heterogeneity. However, because the search has to be continued until the optimum is guaranteed to be found out, this approach gets a scalability classified as −−. For constraint coverage, [25] discusses IT resources and network latency without taking bandwidth and devices' properties into account.

An exhaustive algorithm based on backtrack search is proposed in [6]. An example of the search process for placing two components in two devices is given in Figure 3.1. This algorithm finds out all valid (*i.e.,* conforming to constraints) placements, among which the best one (regarding the objective function) is returned. Such a best placement among all valid ones must be

the optimum, thus [6]'s result quality is classified as ++. However, this algorithm has to traverse the search space, which result in a scalability classified as −−. Dealing with IoT applications' placement in the fog, [6] supports all constraints, location dependency, and heterogeneity.

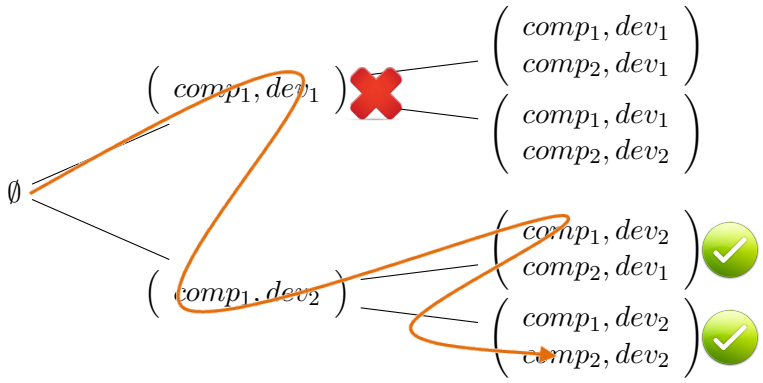


Figure 3.1: Exhaustive Search Example (Exhaustive begins by testing to place $comp_1$ in $dev_1$, which fails because of constraint violations; after placing $comp_1$ in $dev_2$, two valid placements are found out; finally, between these two valid placements, Exhaustive selects and returns the one that better fits the optimization objective.).

Given a placement problem, an exact algorithm guarantees to deliver the optimal result. However, a placement problem's search space exponentially increases with the problem size (*i.e.,* the number of components to place and the number of devices in the infrastructure). With a guarantee of finding the optimal placement, given a large-scale placement problem, both exhaustive algorithm and ILP have to visit a huge number of placements, which results in high execution times and make these algorithms hardly scalable.

### 3.2.2 Metaheuristics

Given a placement problem, metaheuristics visit only a subset of placements in the search space, and return the best placement among visited ones [30, 31]. By making such a trade-off of result quality and scalability, metaheuristics accelerate the search at a cost of losing the guarantee of finding the optimal placement. Each metaheuristic has a specific strategy to get close to the optimum as fast as possible, and stops when the result quality can no longer be improved (or reaching a predefined timeout / maximal iteration round).

[24] iteratively improves a placement $p$ based on hill climbing. At each iteration, it tests all *neighbors*[1] of $p$, and updates $p$ as the best placement among tested ones according to the objective function. In this way, $p$ fits

[1] A placement $p$'s *neighbor* is a placement that changes at most a predefined number of components' hosts based on $p$.

better and better the optimization objective. However, testing all neighbors at each iteration, this approach can repeatedly visit and test some placements, which lowers its efficiency. The search stops until $p$ does not have a better neighbor, and thus this approach's scalability is classified as $-$. Moreover, starting from a random initial placement, the search usually traps into a local optimum, which makes [24]'s result quality classified as $-$. Dealing with IoT applications in the cloud and edge, this approach supports both location dependency and heterogeneity. For constraint coverage, IT resources, network latency, and bandwidth are taken into account, and only devices' properties is not discussed.

Based on simulated annealing, [32] tries to widely scatter visited placements at the beginning of a search. Then, it continuously improves a placement when approaching the end of the search (*i.e.,* getting close to the timeout), and finally returns the best placement among visited ones. Such a procedure is based on a changing probability *proba*, which decreases along with the algorithm execution. At each iteration, a random neighbor $n$ of the current placement $p$ is visited, and $p$ is updated to be $n$ (or not) according to the following principles:

- if $n$ fits better the objective function than $p$, $p$ is assigned to $n$;
- otherwise, if a random value in $[0,\ 1)$ is lower than *proba*, a lower qualified $n$ is also accepted and used to update $p$;
- otherwise, $p$ is not updated, and thus the search continues to visit the next neighbor based on a same $p$.

Different from hill climbing, whose current placement is always the best, [32] allows $p$ being assigned to a placement with lower quality. Along with the search, the value of *proba* decreases continuously. Consequently, $p$ is more and more probable to be improved (rather than worsen) at each iteration, which avoids continuously visiting worse placements when getting close to the end of the search. Such an approach allows escaping from a local optimum, and thus results in a better result quality classified as $+$. During simulated annealing's search, one trajectory (or a set of placements) can be visited multiple times, which makes the algorithm inefficient. Because the search is stopped when a timeout is exceeded or $p$ is no longer significantly improved, [32]'s scalability is classified as $-$. Dealing with applications' placement in the cloud, this approach supports heterogeneous DCs. However, location dependency is not considered, and only the constraint of IT resources is discussed.

Tabu search [33] avoids repeatedly visiting placements by using a tabu list. A tabu list has a given size, and is used to store recently visited placements. At each iteration, based on the current placement $p$, tabu search generates a set of neighbors outside the tabu list. Then, among generated neighbors, the best one is selected to update $p$. By keeping exploring unvisited placements, tabu search approaches the optimal result little by little.

Based on tabu search, this approach can also escape from local optimums, and thus the result quality is classified as +. [33] stops the search when exceeding a timeout or no longer significantly improving $p$, thus the scalability is classified as −. [33] deals with the placement problem in the context of cloud, and supports heterogeneous DCs. However, location dependency is not considered, and only the constraint of IT resources is discussed.

Based on Genetic Algorithm (GA), [26] iteratively refines a population (*i.e.,* a number of placements). At each iteration, GA generates offspring (*i.e.,* new placements) by swapping component-device mappings[2] between parents (*i.e.,* placements of the population). Then, newly generated offspring are added to and thereby enlarges the population. According to the optimization objective, only the best $N$ (*i.e.,* a predefined population size ceiling) placements are kept in the population for further offspring generation. After a certain rounds of iterations, the best placement in the final population is returned. Similar to metaheuristics discussed above, [26] gets closer to the optimum iteration by iteration, and stops the search when exceeding a timeout or no longer significantly improving the best placement. Thus, its result quality is classified as +, and the scalability is classified as −. Dealing with IoT applications in the cloud and edge, this approach supports both location dependency and heterogeneity. For constraint coverage, [26] discusses IT resources and network latency without taking bandwidth and devices' properties into account.

[34] makes placement decisions using Ant Colony Optimization (ACO), which iteratively generates placements according to the probability of each component-device mapping (*i.e.,* the probability of placing a component in a device). At each iteration, such probabilities are tuned according to the evaluation of generated placements' quality, which help to generate better placements according to the optimization objective. When a predefined timeout is exceeded (or when the search no longer significantly improves the placement), the best placement among generated ones is returned. Thus, same as metaheuristics discussed above, this approach's result quality is classified as +, and the scalability is classified as −. Dealing with applications' placement in the cloud, this approach does not support location dependency, and considers the constraints of IT resources and bandwidth consumption. To respect each link's bandwidth capacity, [34] limits each device's I/O throughput. Such an approach must be based on a homogeneous network (*i.e.,* each device is connected by a single network link), and does not suit the heterogeneous fog (*i.e.,* a device can be connected by multiple links, and using the sum of these links' capacities as the device's limit of I/O throughput still can exceed certain links' bandwidth capacities). As a result, the heterogeneity is not supported.

Among discussed metaheuristics, hill climbing, simulated annealing, and

[2] A component-device mapping indicates the component's host (*i.e.,* the device).

tabu search update a placement by visiting and testing its neighbors. Given a problem, they follow one specific search trajectory for improving the result quality little by little. Differently, GA maintains a population, and ACO maintains a set of probabilities. Both GA and ACO make it possible to "jump" (rather than improving little by little) to the optimum. However, such a jump is in a probabilistic manner without guaranteeing the improvement efficiency. Moreover, all these algorithms start the search from random initial placement(s), because of which the efficiency can not be guaranteed. Furthermore, these algorithms' result quality relies on predefined parameters (*e.g.,* tabu list size, population size, timeout, etc), whose proper values are problem-dependent and difficult to assign without experience.

### 3.2.3 Heuristics

Both exact algorithms and metaheuristics can be adapted to deal with a wide range of optimization / search problems. Differently, a heuristic is specially designed for a specific kind of problems, so that it can make use of specific features of its problems to improve the algorithm efficiency. Same as metaheuristics, heuristics do not guarantee to find optimal results. For solving placement problems, some heuristics (such as [35], [36], and [28]) rely on human knowledge (*i.e.,* the experience of what kind of placements can be high-qualified) to guide the search to a satisfactory result. Considering components' concurrency to limited resources in placement problems, some heuristics (such as [27] and [37]) prioritize components related to resource requirements hard to satisfy, which helps to find a valid placement rapidly.

[35] selects DCs of distributed clouds for hosting an application dealing with end users' requests. In order to optimize such an application's performance, [35] tries to minimize two values: i) selected DCs' distance (*i.e.,* network latency or hop number) to the end users, ii) selected DCs' *diameter* (*i.e.,* the longest distance between selected DCs). To addresses this problem, the heuristic proposed in [35] first selects the nearest DC to the end users' center. Then, as long as selected DCs do not have enough resources to fulfill the application, a new DC that minimizes the increase of diameter is selected. In this way, DCs are selected until sufficient resources can be used to host the application while fitting the optimization objective. This heuristic guides the search to a satisfactory placement close to the optimum, which makes both result quality and scalability of this approach classified as +. Although the heuristic fits well this approach's optimization objective, the proposed model does not consider the communication between components. Even if selected DCs' diameter is minimized, [35] can place components communicating with each other in different DCs while placing components without inter-communication in a same DC, which lowers the application's performance. Without discussing the communication, the constraint of network resources is not covered, neither is the constraint of devices' properties.

Heterogeneous DCs can be supported, and the considered application is tied to end users' locations. However, this approach uses one location (*i.e.,* end user locations' center) to position all end users, which must be based on the assumption that all end users are in a same localization area (otherwise, the application will be placed around the center that is not close to any end user). As a result, [35] suits only applications tied to a single location, and can not deal with applications serving multiple localization areas.

To overcome [35]'s limitation—an application can be tied to only one location, instead of localizing an application (*i.e.,* guiding an application to be close to its end users) as a whole, [36] localizes each component with the following heuristic. By using geographic coordinates to position DCs, end users, and components, each component is initially assigned to the geographic center of end users it communicates with. Then, through an iterative algorithm, each component's coordinates is updated iteratively to the center of end users and components it communicates with. Finally, each component is placed in its nearest DC with enough resources to host it. Based on this heuristic, each component $c$ can be placed in a DC close to end users and components that $c$ communicates with. Similar to [35], both result quality and scalability of [36] are classified as +, and the constraints of network resources and devices' properties are not discussed. [36] assumes that the network latency between two devices is proportional to their geographical distance. Such an assumption suits distributed clouds with homogeneous DCs, but not the fog with heterogeneous devices. In the fog, the network latency between devices connected to different sub-networks (*e.g.,* two mobiles relatively connected to WiFi and 4G networks) can be relatively high, even if these devices are close to each other geographically, and thus the assumption that geographical proximity leads to a low network latency is not always true in the fog. As a result, the heterogeneity is considered to be not supported.

In an infrastructure composed of distributed devices, [28] places a Data Stream Processing application processing data generated by certain devices. By defining a link's usage as the product of its latency and consumed bandwidth, [28] aims at minimizing the global network usage (*i.e.,* the sum of each link's usage). A latency space is used to position devices and components, in which the distance between two devices represents (and is proportional to) the network latency between them. After constructing the latency space with Vivaldi algorithm [38], the following heuristic is launched to make placement decisions. Each component's coordinate (in the latency space) is initialized according to coordinates of devices providing data to this component. Then, each component's coordinate is refined by an iterative algorithm to better fit the optimization objective. Finally, each component is placed in the nearest device respecting all considered constraints. This approach guides each component to its optimal coordinate for minimizing the network usage, makes its result quality and scalability both classified as +, and sup-

ports both location dependency and heterogeneity. However, a device closer to the optimal coordinate is not guaranteed to fit better the optimization objective (compared with another device). Moreover, this approach does not consider the constraints of bandwidth and devices' properties.

For placing IoT applications in the fog, [27] takes all constraints in the constraint coverage criterion (*i.e.,* IT resource, network resource, devices' properties) into account. Given a component $c$, a device is *compatible* to $c$ if it has enough resources and all properties (*e.g.,* OS, installed software packages) required by $c$[3]. Before placing any component, [27] counts each component's compatible devices. Regarding that a component with more compatible devices is likely to have more choices to be placed, [27] places priorly a component with less compatible devices, which helps to avoid resources that must be provided to certain components being offered to other ones. By applying such a heuristic specialized for accelerating the search, this approach's scalability is classified as $++$, and both location dependency and heterogeneity are supported. However, this approach returns the first valid placement found, whose quality is rather random. Thus, [27]'s result quality is classified as $--$. Furthermore, during the search, in which components are placed one by one, the actual number of compatible devices of each component evolves (because of resource consumption). Consequently, this heuristic is prone to encounter counter-examples, which can incur high execution times.

For overcoming [27] drawback—evaluating resource scarcity statically, [37] orders components with resources' evolution taken into account during the search. Each time when a component is placed, [37] re-evaluates the resource scarcity for each component, and satisfies first a component requiring a higher ratio of its accessible resources. Given a set of components, a "larger" component with respect to available resources is given a higher priority. This heuristic is specialized for accelerating the search without taking result quality into account. Thus, this approach's scalability is classified as $++$, and the result quality is classified as $--$. Dealing with the placement problem in the context of cloud, this approach supports heterogeneous DCs, but the location dependency is not discussed. Moreover, [37] does not take network resources and devices' properties into account. Using only CPU and RAM requirements as components' ordering criteria, this heuristic does not suit the fog containing resource-constrained links.

To deal with large-scale problems, [39] proposes a hierarchical distributed placement decision-making system, which consists of a Root Level Manager (RLM), Mid Level Managers (MLM), and Low Level Managers (LLM). Each placement request is sent to the RLM, then forwarded to a MLM, and finally forwarded to a LLM. A LLM covers a set of devices, and verifies if

[3]Placing components in their compatible devices do not guarantee to satisfy required network resources.

these devices have enough resources to host the received application. When a placement request can not be fulfilled by a LLM, this request is sent back to the MLM, and forwarded to another LLM (the same policy is applied between MLMs and the RLM). Thanks to decentralized managers, the placement decision-making can be distributed, which makes this mechanism more scalable. Hence, this approach's scalability is classified as ++. However, because LLMs can not coordinate with each other, this approach does not suit widely distributed applications that must be managed by multiple LLMs. Moreover, each LLM makes placement decisions based on its covered devices' information. Without a global view of the infrastructure, this approach is prone to trap into local optimums, and thus the result quality is classified as −. Dealing with applications' placement in a cloud DC, [39] supports heterogeneous hosts, and considers limits of network latency and bandwidth between hosts in the DC. However, location dependency is not supported in this approach, and the constraint of devices' properties is not discussed.

Among discussed heuristics, [35], [36], and [28] are proposed for getting results with satisfactory quality. However, given a large number of applications / components to place in an infrastructure with resource-constrained devices / links, these approaches can cause high execution times (which can be weeks or years). On the other hand, [27], [37], and [39] are proposed for accelerating the search. Nevertheless, the result quality is not (or weakly) taken care of. To sum up, none of these works simultaneously takes result quality and scalability into account.

### 3.2.4 Comparison and Summary

According to the criteria introduced in Section 3.1, related works are compared and summarized in Table 3.1.

| | | Searching Approach | Constraint Coverage | | Location Dependency | Device Heterogeneity | Result Quality | Scalability |
|---|---|---|---|---|---|---|---|---|
| | | | Network Latency | Bandwidth | | | | |
| exact | [25] | ILP | ✓ | ✗ | ✓ | ✓ | ++ | −− |
| | [6] | exhaustive | ✓ | ✓ | ✓ | ✓ | ++ | −− |
| metaheuristic | [24] | hill climbing | ✓ | ✓ | ✓ | ✓ | − | − |
| | [32] | simulated annealing | ✗ | ✗ | ✗ | ✓ | + | − |
| | [33] | tabu search | ✗ | ✗ | ✗ | ✓ | + | − |
| | [26] | GA | ✓ | ✗ | ✓ | ✓ | + | − |
| | [34] | ACO | ✗ | ✓ | ✗ | ✗ | + | − |
| heuristic | [35] | device ordering | ✗ | ✗ | −[4] | ✓ | + | + |
| | [36] | device ordering | ✗ | ✗ | ✓ | ✗ | + | + |
| | [28] | device ordering | ✓ | ✗ | ✓ | ✓ | + | + |
| | [27] | component ordering | ✓ | ✓ | ✓ | ✓ | −− | ++ |
| | [37] | component ordering | ✗ | ✗ | ✗ | ✓ | −− | ++ |
| | [39] | hierarchical | ✓ | ✓ | ✗ | ✓ | − | ++ |

Table 3.1: Initial Placement Related Work Summary.

Table 3.1 gives studied works' coverage of network resource constraints. For other constraints, all studied works take IT resources (*i.e.,* CPU, RAM, Disk[5]) into account, and only [6] and [27] consider devices' properties.

Because exact algorithms / metaheuristics are highly generic, even though some exact algorithms / metaheuristics listed in Table 3.1 do not consider all criteria, they can be adapted to do so. Differently, a heuristic is specialized for the problem it deals with, which can be difficult to generalize.

This work takes all aforementioned criteria into account, finds out results close to the optimum (*i.e.,* with result quality classified as +), and uses approaches specifically for accelerating the search (*i.e.,* with scalability classified as ++). More details of this work's proposition are given in Chapter 4.

[4] [35] uses a single location to localize considered applications. Such an approach does not fit applications tied to multiple localization areas. Thus, the location dependency is considered to be partially supported.

[5] Some related works consider only one or two kinds of IT resources. However, because of IT resources' similarity in placement problems, these works can be easily adapted to deal with all IT resources.

# 4

# Proposition for Initial Placement

Contents



As stated in Chapter 1, this work deals with the problem of placing IoT applications in the fog, and aims at optimizing placed applications' performance. This chapter proposes our approach for solving the initial placement problem (*i.e.,* to make initial placement decisions when no application is placed), and is organized as follows. Section 4.1 formulates the initial placement problem with a model and an objective function. Section 4.2 gives a naive algorithm—FirstFit for making initial placement decisions. Based on FirstFit, five heuristics are proposed in Section 4.3. The complexity of proposed heuristics are analyzed and compared with FirstFit in Section 4.4. Finally, Section 4.5 summarizes this chapter.

## 4.1 Initial Placement Problem Formulation

### 4.1.1 Model

A placement problem consists of: i) an infrastructure, which provides resources to host applications, ii) a set of applications to place, which require and consume resources provided by the infrastructure.

A fog infrastructure contains two kinds of devices: i) *fog nodes* (*e.g.,* cloud, edge server), which provide processing and storage resources; ii) *appliances* (*e.g.,* sensor, actuator), which provide sensing / actuating services. Only fog nodes can host applications. An infrastructure also consists of links, which connect devices and provide network resources.

An IoT application is composed of components, bindings, and appliances[1]. A *component* is a software element that can be executed on one fog node. A *binding* is a communication channel that connects a couple of components or a component and an appliance.

An infrastructure (*resp.,* application) is modeled as a graph. Each vertex is a device (*resp.,* a component or an appliance). Each edge is a link (*resp.,* binding). The model is defined and summarized in Table 4.1.

| | |
|---|---|
| $Infra$ | a fog infrastructure |
| $node_i$ | a fog node of $Infra$ |
| $node_i.CPU$ | $node_i$'s available CPU capacity |
| $node_i.RAM$ | $node_i$'s available RAM capacity |
| $node_i.DISK$ | $node_i$'s available DISK capacity |
| $appliance_i$ | an appliance of $Infra$ |
| $link_i$ | a link of $Infra$ |
| $link_i.LAT$ | $link_i$'s network latency |
| $link_i.BW$ | $link_i$'s available bandwidth capacity |
| $Apps$ | a set of applications to place |
| $comp_i$ | a component of an application in $Apps$ |
| $comp_i.ReqCPU$ | $comp_i$'s CPU requirement |
| $comp_i.ReqRAM$ | $comp_i$'s RAM requirement |
| $comp_i.ReqDISK$ | $comp_i$'s DISK requirement |
| $comp_i.DZ$ | $comp_i$'s Dedicated Zone |
| $bind_i$ | a binding of an application in $Apps$ |
| $bind_i.ReqLAT$ | $bind_i$'s latency requirement |
| $bind_i.ReqBW$ | $bind_i$'s bandwidth requirement |
| $app_i$ | an application in $Apps$ |
| $app_i.components$ | all components of $app_i$ |
| $app_i.appliances$ | all appliances of $app_i$ |

Table 4.1: Summary of Notations.

The proposed model characterizes each component *comp* with: i) $ReqCPU$, $ReqRAM$, and $ReqDISK$, which respectively indicate CPU, RAM, and DISK capacities that *comp* needs; ii) a *Dedicated Zone* (DZ), which is a deployment area composed of a set of fog nodes respecting *comp*'s requirements on fog node properties (*e.g.,* privacy, OS, etc). Bindings are characterized with requirements as well. $bind_i.ReqBW$ designates $bind_i$'s bandwidth requirement. $bind_i.ReqLAT$ indicates the maximal network latency that $bind_i$ can accept. Components and appliances of an application $app_i$ are denoted by $app_i.components$ and $app_i.appliances$, respectively.

[1]Considering that a sensing / actuating service is tied to and must be executed on its appliance, both services and hardware of appliances are named "appliance" in this work.

A *placement* maps each component onto a fog node, as in the following example:

$$\begin{pmatrix} comp_1, node_i \\ comp_2, node_j \\ \dots \\ comp_n, node_k \end{pmatrix}$$

A *solution* to a placement problem is a placement that satisfies the following constraints:

i) each component is placed in its DZ;
ii) each fog node's CPU / RAM / DISK consumption does not exceed the fog node's capacity;
iii) each link's bandwidth consumption does not exceed the link's capacity;
iv) each binding's latency does not exceed the binding's requirement.

### 4.1.2 Objective Function—Weighted Average Latency

As aforementioned, this work aims at optimizing applications' performance, namely, minimizing applications' response times. A request's response time is composed of communication time (*i.e.,* time spent to transfer messages) and execution time (*i.e.,* time spent within components for processing). With processing resource requirements predefined for each component, the execution time is assumed to change insignificantly with placement, and this work focuses on minimizing the communication time.

A placement problem can have multiple solutions, among which only one solution can be selected as the placement decision. To select the solution which helps to minimize applications' response times (or requests' communication times), the following objective function is applied:

$$min : WAL = \sum_{bind \in Apps} \frac{bind.ReqBW}{total\_BW} \times bind.Lat$$

$$total\_BW = \sum_{bind \in Apps} bind.ReqBW$$

$total\_BW$ is the total bandwidth requirement of all the bindings. $bind.Lat$ is $bind$'s latency regarding the evaluated solution[2]. The objective function is to minimize *Weighted Average Latency* (WAL) of *Apps*. Considering that a binding with a high $ReqBW$ can strongly impact an application's response time, each binding's latency (*i.e.,* bind.Lat) is weighted by a proportion of its $ReqBW$ regarding $total\_BW$. Through minimizing WAL,

[2]Given a solution in which each component is placed in a fog node, each binding must be correspondingly placed in a communication path composed of a set of links. A *binding's latency* is the network latency of its communication path.

latencies of bindings, especially of the bindings with high $ReqBW$, are minimized, which helps to reduce applications' response times. Ideally, given a placement problem, the solution with minimal WAL should be selected and returned.

## 4.2 FirstFit Search—A Naive Approach

Given a placement problem with $m$ fog nodes and $n$ components, there exists $m^n$ possible placements, which compose this problem's search space. Such a search space can be represented using a tree data structure, in which each placement is a leaf whose depth is $n$. Any branch from the tree root to a leaf builds a placement by mapping successively each component to a fog node. An example of this data structure is given in Figure 4.1.

Based on depth-first search, a FirstFit backtrack algorithm is implemented to find solutions in such trees. As named, FirstFit returns the first solution found. It deals with a set of applications: components of all considered applications are mixed and placed one by one. When FirstFit tries to place a component *comp*, all kinds of constraints are verified for *comp* and components already placed. As depicted in Figure 4.1, when a constraint verification is passed, FirstFit continues with the next component. Once all the components are successfully placed, it implies that a solution is found. If a constraint verification fails, FirstFit tests the next fog node to place *comp*. If all possible fog nodes are tested, and no suitable fog node is found, FirstFit backtracks to the previous component to test other possibilities (*i.e.,* change the previous component's host). When FirstFit backtracks from the first component, it implies that the search space has been traversed, and no solution exists. By continuing the search until a solution is found or the search space is traversed (when no solution exists), FirstFit guarantees to find a solution, if any exists.

A number of tests must be carried out before FirstFit returns (*e.g.,* FirstFit does five tests in Figure 4.1). To reduce the number of tests (*i.e.,* to accelerate the search), when trying to place a component *comp*, fog nodes out of *comp*'s DZ are not tested (because a component must be placed in its DZ).

When FirstFit tries to place a component, fog nodes are tested one by one, which implies an order of fog nodes. As a naive approach, FirstFit tests the fog nodes in a random order for each component. Similarly, components are also ordered randomly to be placed one after another. As a result, FirstFit has no guarantee on WAL values of returned solutions or needed numbers of tests, which can incur low result quality and high execution times.

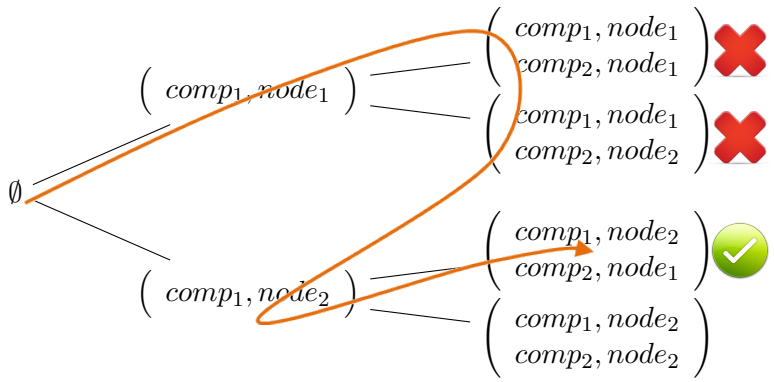


Figure 4.1: FirstFit Search Process Example (FirstFit begins by testing to place $comp_1$ in $node_1$, which passes; then it fails to place $comp_2$ in $node_1$ and $node_2$ because of constraint violations; having tested all possible fog nodes for $comp_2$, FirstFit backtracks to $comp_1$ and continues the search; finally, it returns the first solution found, in which $comp_1$ and $comp_2$ are respectively placed in $node_2$ and $node_1$.).

## 4.3 Initial Placement Heuristics

To overcome FirstFit's drawbacks, Section 4.3.1 presents two heuristics based on the manipulation of fog nodes' order, and Section 4.3.2 proposes the other two heuristics manipulating components' order. Section 4.3.3 presents the last heuristic, which avoids meaningless tests of fog nodes. These heuristics' combination is discussed in Section 4.3.4.

### 4.3.1 Fog Nodes Ordering-Based Heuristics

For FirstFit, different fog node orders can result in different solutions and different numbers of tests. For example, given the placement problem in Figure 4.1, if $node_2$ is tested first, as depicted in Figure 4.2, another solution is found with only two tests.

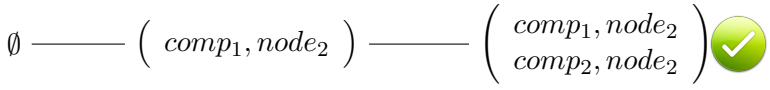


Figure 4.2: Fog Node Order Impact.

To improve FirstFit, this subsection introduces two heuristics—Anchor-based Fog Nodes Ordering (AFNO), which takes care of fog nodes' initial order for each component, and Dynamic Anchor-based Fog Nodes Ordering (DAFNO), which dynamically update fog node orders during the search.

### Anchor-based Fog Nodes Ordering

Containing fog nodes in different network positions, a fog infrastructure has the potential to localize sensory data's processing. To take this advantage, given a component *comp* to place, fog nodes must be tested from local ones (*i.e.,* fog nodes close to components and appliances *comp* communicates with) to remote ones. More precisely, a fog node better fitting our objective function—minimizing WAL should be given a higher priority to be tested.

By defining a component $c$'s *anchor* as the fog node on the barycenter of $c$'s network communication (*i.e.,* the fog node that minimizes WAL for hosting $c$ without considering resource limits[3]), we use each component *comp*'s anchor to decide fog nodes' testing order for placing *comp*. With fewer constraints taken into account, anchors' calculation has a lower complexity than the placement problem. Given an infrastructure model $infra$ and an application model *app* as inputs, Algorithm 1 calculates anchors for *app*'s components.

**Algorithm 1:** getAnchors

```
   Input: infra, app
1  center ← infra.centerNode( app.appliances() );
2  for each comp ∈ app.components() do
3      ancs[comp] ← comp.closestNodeInDZ( center );
4  compList ← app.components();
5  while compList ≠ ∅ do
6      comp ← compList.firstElement();
7      compList.remove(comp);
8      newAnc ← calculateAnc( comp, ancs, infra );
9      if ancs[comp] ≠ newAnc then
10         ancs[comp] ← newAnc;
11         compList ← compList ∪ comp.boundComps();
12 return ancs;
```

In Algorithm 1, line 1–3 globally localizes *app* based on network positions of *app*'s appliances. In line 1, by calling $infra.centerNode()$, *center* is assigned to the fog node minimizing average network latency to *app*'s appliances returned by $app.appliances()$. In line 2, $app.components()$ returns *app*'s components. In line 3, $comp.closestNodeInDZ()$ returns the fog node that minimizes network latency to *center* in *comp*'s DZ. Through line 1–3, each component's anchor is assigned to the nearest fog node to *center* in its DZ.

Line 4–11 further localizes each component. Being initialized to contain

[3]Constraints ii, iii, and iv stated in Section 4.1 are not considered when calculating anchors. However, constraint i—placing each component in its DZ must be conformed.

all the components of *app* (line 4), *compList* stores the components to iteratively get anchor-update tests. At each iteration (line 6–11), one component *comp* is selected and removed from *compList* (line 6–7), and then tested to update its anchor (line 8–10). When *comp*'s anchor is updated, its bound components (*i.e.,* components that *comp* communicates with) can change with it, hence they are added into *compList* again to get tests later (line 11). Once *compList* is empty, the calculation of anchors finishes and the algorithm returns (line 12).

According to the definition of WAL (see Section 4.1), for a component *comp*, if a new anchor leading to a smaller WAL exists, the new anchor must be closer to one of the devices *comp* communicates with. Therefore, in each test to update an anchor (line 8), *calculateAnc*() evaluates fog nodes between *comp* and *comp*'s bound components (*i.e.,* fog nodes on communication paths from *comp*'s current anchor to bound components' anchors) and between *comp* and *comp*'s bound appliances. Finally, *calculateAnc*() returns the fog node that minimizes WAL among evaluated ones.

With anchors calculated by Algorithm 1, before testing fog nodes to place a component *comp*, AFNO sorts fog nodes in ascending order of network latency to *comp*'s anchor. Therefore, the first solution found must be close to the anchors and thus helps to minimize WAL. Moreover, by minimizing WAL, anchors help to minimize bindings' latencies, which makes bindings' requirements on maximal latency prone to be satisfied. Thus, AFNO also guides components to network positions close to a solution, and thereby accelerates the search.

AFNO aims at improving FirstFit on both WAL and execution time. Its functionality is evaluated in Chapter 5.

### Dynamic Anchor-based Fog Nodes Ordering

Regarding anchor's definition, a component *comp*'s anchor depends on locations of components and appliances that *comp* communicates with. When AFNO calculates *comp*'s anchor, other components are considered to be located on their anchors. However, during the search, in which all constraints must be conformed, a component's host can be different from its anchor, which can make other anchors outdated. With outdated anchors, the search is no longer guided properly. In order to guide the search with up-to-date anchors, Dynamic Anchor-based Fog Nodes Ordering (DAFNO) extends AFNO by updating anchors dynamically.

During a search, each time when a component is placed in a fog node other than its anchor, DAFNO updates anchors with Algorithm 2. Algorithm 2's inputs contain an infrastructure model $infra$, a table *ancs* that stores the anchors, the component that is not placed in its anchor *placedComp*, and the model of *placedComp*'s application *app*.

Same as Algorithm 1, Algorithm 2 updates anchors of components in

**Algorithm 2:** updateAnchors

```
   Input: infra, app, ancs, placedComp
1  compList ← placedComp.boundComps();
2  while compList ≠ ∅ do
3      comp ← compList.firstElement();
4      compList.remove(comp);
5      if not comp.isPlaced() then
6          newAnc ← recalculateAnc( infra, app, ancs, comp );
7          if ancs[comp] ≠ newAnc then
8              ancs[comp] ← newAnc;
9              compList ← compList ∪ comp.boundComps();
```

*compList* one by one until no further update is possible (*i.e.,* when *compList* is empty). In line 1, *compList*, which stores components to get anchor-update tests, is initialized as *placedComp*'s bound components (*i.e.,* components that *placedComp* communicates with). Because anchor-updates only help for components not placed yet, only if *comp* is not placed (line 5), *comp* is tested to update its anchor (line 6–9). In each test to update an anchor (line 6), *recalculateAnc*() evaluates fog nodes between *comp* and *comp*'s bound components[4] and between *comp* and *comp*'s bound appliances. Finally, *recalculateAnc*() returns the fog node that minimizes WAL[5] among evaluated ones.

With Algorithm 2, DAFNO dynamically sorts fog nodes according to dynamically updated anchors. Impacts of anchors with / without dynamic updates are compared in Chapter 5.

### 4.3.2 Components Ordering-Based Heuristics

The other order in FirstFit, components' order, can also impact the search's speed and result, as depicted in the following two examples.

Given the placement problem in Figure 4.1, if $comp_2$ is placed first, as in Figure 4.3, only three tests are needed.

Besides the impact on number of tests, different component orders can also lead to different solutions. Consider another example of placing two components $comp_1$ and $comp_2$ in two fog nodes $node_1$ and $node_2$, in which only one component can be placed in $node_1$ (because of $node_1$'s capacity

[4]For each component *boundComp* bound with *comp*, fog nodes on the communication path between *comp*'s current anchor and *boundComp*' host (if *boundComp* is placed) or *boundComp*' anchor (if *boundComp* is not placed) are evaluated.

[5]When calculating WAL, a placed component is located on its host, and a component not placed yet is located on its anchor.

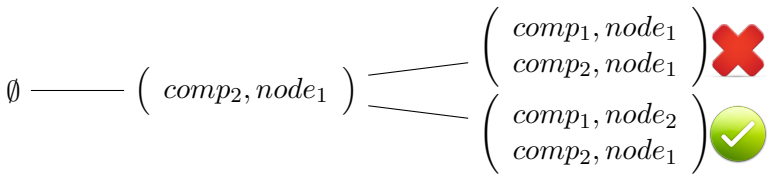


Figure 4.3: Component Order's Impact on Number of Tests.

limit). If $comp_2$ is placed first, the search process is same to Figure 4.3. If $comp_1$ is placed first, as in Figure 4.4, it results in another solution with a different WAL.

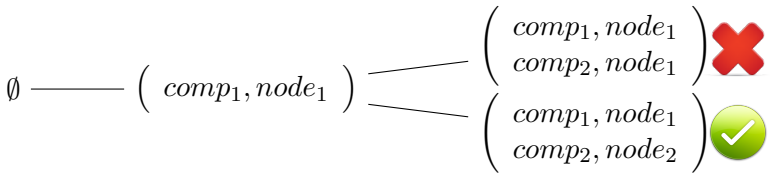


Figure 4.4: Component Order's Impact on WAL.

To further improve FirstFit, this subsection introduces two heuristics—Initial Component Ordering (InitCO), which is responsible for the initial component order, and Dynamic Components Ordering (DCO), which dynamically update components' order during the search.

### Initial Component Ordering

With AFNO / DAFNO, multiple components can target to a same anchor, which leads to components' concurrence to limited resources. To better fit the objective function, a component that stronger impacts WAL should be given a higher priority (*i.e.,* be placed priorly).

Consider a component connected by many bindings requiring high bandwidths, if it is placed far from its anchor, WAL can increase significantly. With this consideration, a *component's bandwidth requirement* is used to measure its impact on WAL, which is defined as the sum of bandwidths required by bindings connecting this component to other components / appliances. Therefore, components are sorted in descending order of their bandwidth requirements, and then placed one by one.

InitCO purposes to lower WAL and to speed up the search. Algorithms with / without InitCO are compared in Chapter 5.

### Dynamic Components Ordering

Because of components' / bindings' concurrence to limited resources and bindings' maximal latency, placing a component depends on placed ones, and different component orders can result in different numbers of tests.

Numbers of tests carried out by searches with and without backtrack can have a huge difference. For example, given $n$ components ordered as $\prec c_1, c_2, \ldots, c_i, \ldots, c_j, \ldots, c_n \succ$, if former placed $c_i$ makes it impossible to place $c_j$, after finding out that no fog node suits $c_j$, FirstFit has to backtrack from $c_j$ until $c_i$. Without the knowledge that failures of placing $c_j$ concern $c_i$, before arriving at $c_i$, FirstFit must test all the possibilities of placing $\{ c_{i+1}, c_{i+2} \ldots c_j \}$, which leads to $\|c_{i+1}\| \times \|c_{i+2}\| \times \cdots \times \|c_j\|$ tests in the worst case ($\|c\|$ is the number of fog nodes in $c$'s DZ). Such a huge amount of tests can be avoided if $c_j$ is ordered before $c_i$. Under the new order, $c_j$ is placed without constraints introduced by $c_i$. If $c_i$ still has a suitable fog node, FirstFit can successfully place $c_i$ and $c_j$ without backtrack. When $n$ components are placed without backtrack, FirstFit needs at most $\|c_1\| + \|c_2\| + \cdots + \|c_n\|$ tests.

As fog nodes / links in a fog infrastructure are heterogeneous and can be resource-constrained, it is quite probable to encounter backtracks during a search. A component order that does not lead to any backtrack can highly accelerate the search. However, such an order can not be found before the search, due to the difficulty of predicting resources' evolution (*i.e.,* changes of available resources brought by each component's placing).

By making use of the knowledge obtained during the search, DCO dynamically adjusts components' order to avoid backtracks. Considering that a component can be constrained by components ordered before it, once FirstFit fails to place a component *comp*, instead of backtracking, DCO moves *comp* forward by $compNB \times stepLen$ components. $compNB$ is the number of components ordered before *comp*. $stepLen$ is a predefined ratio, $0 \leq stepLen \leq 1$. DCO with a certain $stepLen$ is denoted by DCO($stepLen$), such as DCO(0), DCO(0.1), ..., DCO(1). *comp* is moved forward by at least one component each time. An example is given in Figure 4.5.

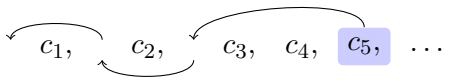


Figure 4.5: Example of Moving Forward a Component (regarding that $c_5$ can not be placed, DCO(0.5) moves it forward to $c_3$'s position. Correspondingly, $c_3$ and $c_4$ are moved backward. Then, FirstFit fails another two times to place $c_5$, and $c_5$ is moved to the head finally.).

Each move produces a new component order, under which the search is continued. Moved components (*e.g.,* $c_3$, $c_4$, and $c_5$ after the first move in Figure 4.5) must be re-tested to get placed, but there is no need to redo the search for not moved components (*e.g.,* $c_1$ and $c_2$ after the first move in Figure 4.5), as their hosts can be reused in the continued search. Different values of $stepLen$ lead to different moves. Consider the two extreme values: given a component *comp* to move forward, i) DCO(0) moves *comp* forward by one component each time. Obtained search results can be reused as

much as possible, while there can be many moves before being able to place *comp*; ii) DCO(1) orders *comp* before components that constrain it with only one move, but obtained search results can not be reused. Similar to the difference between DCO(0) and DCO(1), a lower (*resp.,* higher) *stepLen* leads to more (*resp.,* fewer) moves, but more (*resp.,* less) result reutilization.

To avoid infinite loops, each component order can be tested only once. DCO can move a component forward several times until an untested order is produced. If no new order can be produced, the algorithm backtracks as in FirstFit, so that the possibility to traverse the search space is retained, and the guarantee to find an existing solution is kept.

DCO reorders components to place priorly the ones hard to satisfy, and thereby accelerates the search. The performance of DCO and the influence of *stepLen* are evaluated in Chapter 5.

### 4.3.3 Partial Fog Nodes' Testing-Based Heuristic

With AFNO / DAFNO, fog nodes are tested from local ones to remote ones. When testing to place a component *comp*, fog nodes far from *comp*'s anchor can always violate certain bindings' requirements on maximal latency. To avoid such meaningless tests of fog nodes, the heuristic Latency Failure Cap (FailCap) caps the maximal number of adjacent failures caused by violations of bindings' maximal latencies. FailCap with a predefined cap value $failNB$ is denoted by FailCap($failNB$). An example is given in Figure 4.6.

$$n_1,\ \boxed{n_2},\ n_3,\ \boxed{n_4},\ \boxed{n_5},\ n_6$$

Figure 4.6: Example of FailCap (fog nodes ordered as $\prec n_1, n_2, \ldots, n_6 \succ$ are tested to host a component *comp*. Placing *comp* in each fog node in red exceeds the maximal latency of a certain binding. FailCap(2) stops the test at $n_5$ and concludes "fail to place *comp*" without testing $n_6$, because $n_5$ is the second adjacent failure caused by latency.).

As bindings' maximal latencies are not considered when calculating anchors, a low $failNB$ value risks of missing proper fog nodes (such as $n_6$ in Figure 4.6, which satisfies bindings' maximal latencies). A higher $failNB$ value lowers such risks at a cost of more tests.

To keep the guarantee of finding an existing solution, all fog nodes must be tested in two cases: i) when the algorithm attempts to backtrack; ii) when the algorithm tests to place a component backtracked to. Without DCO, the algorithm always backtracks when it fails to place a component. In this case, FailCap does not make any difference (*i.e.,* can not avoid any tests). Thus, FailCap must be combined with DCO, so that multiple component orders can be tested, and FailCap helps DCO to get a proper component order more rapidly.

### 4.3.4 Heuristics' Combination

Five heuristics are proposed in this section. Each heuristic is designed for certain gains and causes some costs.

Through AFNO, local fog nodes are tested priorly, which leads to lower WALs and lower numbers of tests. However, because of anchors-calculating and fog nodes-ordering, AFNO introduces an overhead to localize each component. DAFNO updates anchors dynamically, which keeps anchors up-to-date. Nevertheless, each update implies some calculation, which is a further overhead. By manipulating another order—components' order, InitCO purposes to lower WAL. As a cost, components must be sorted according to their bandwidth requirements. With DCO, backtracks that incur huge amounts of tests can be avoided. However, for each avoided backtrack, it has to test a set of component orders, and redo the search for certain components. FailCap helps to avoid meaningless tests of fog node, but it risks of missing proper fog nodes, especially when $failNB$ is assigned to a low value.

Besides gains and costs, the heuristics also have different dependencies. AFNO and DCO do not depend on other heuristics. Differently, DAFNO needs anchors being initialized by AFNO. InitCO must be based on AFNO / DAFNO, otherwise, ordering components while placing them in random fog nodes does not make sense. FailCap works only when i) local fog nodes are tested priorly; ii) new component orders can be tested. Hence, FailCap depends on both AFNO / DAFNO and DCO.

All the heuristics can be combined. The search process with all the heuristics applied is depicted in Figure 4.7.

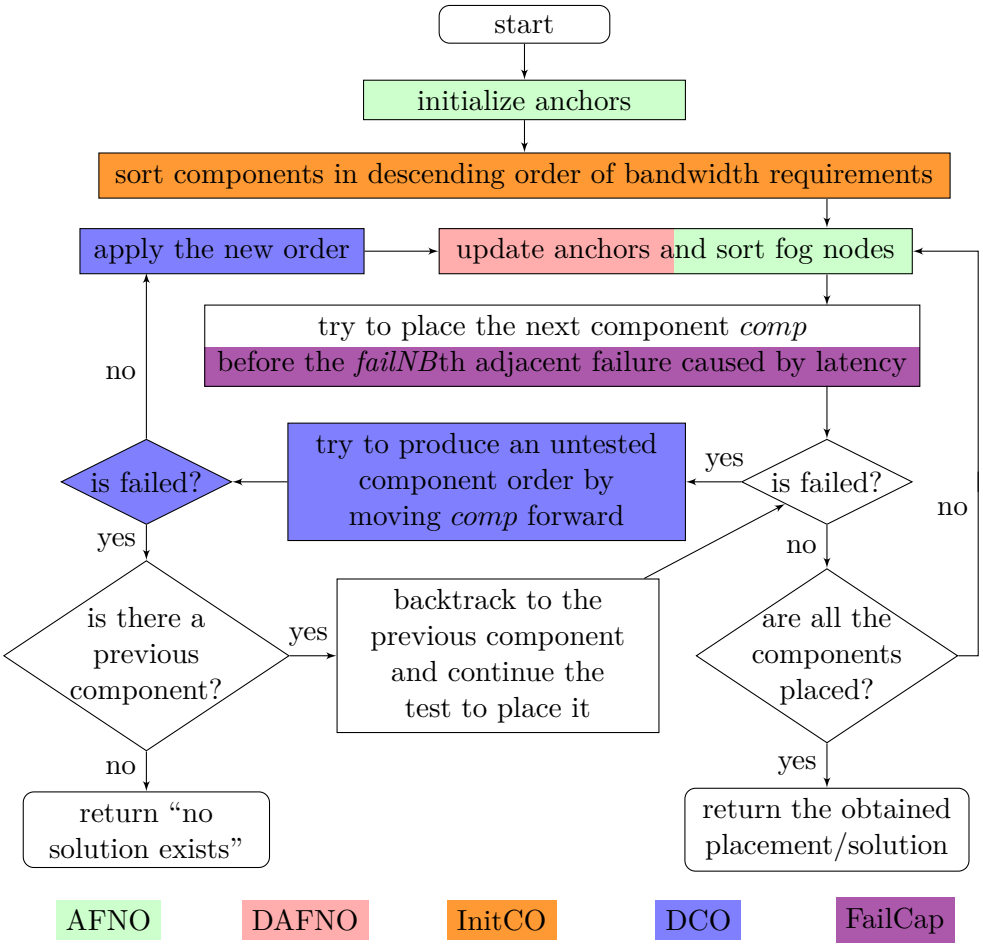


Figure 4.7: Search Process with Combined Heuristics[6](each color indicates phases of a heuristic, uncolored phases are of FirstFit).

The combination of these heuristics should make the placement approach more scalable, and lowers applications' response times.

## 4.4 Initial Placement Algorithms' Complexity

Section 4.3 proposes a heuristic algorithm, which combines and applies five heuristics based on FirstFit. This section first presents an example to discuss a heuristic search's worst case, then gives the proposed algorithm's complexity, and finally analyses each heuristic's impacts on the complexity. As the search process of the heuristic algorithm can be complicated by *stepLen* value lower than 1 (*stepLen* is the parameter of the heuristic DCO,

[6]For simplification, Figure 4.7 does not show that, when the algorithm attempts to backtrack (*i.e.,* DCO fails to produce an untested component order), all fog nodes are tested without taking FailCap into account.

see Section 4.3.2), for simplicity, the impact of *stepLen* is discussed after a first discussion with $stepLen = 1$. If not specified, $stepLen = 1$ in this section.

For placing three components $c_1$, $c_2$, and $c_3$ in an infrastructure with two fog nodes $n_1$ and $n_2$, the following two situations are not valid: i) placing $c_2$ and $c_3$ in $n_1$ simultaneously, which exceeds $n_1$'s capacity (*i.e.,* CPU, RAM, DISK); ii) separating $c_2$ and $c_3$ in two fog nodes, which violates requirements (*i.e.,* latency, bandwidth) of bindings between $c_2$ and $c_3$. Thus, there exist only two solutions: $\begin{pmatrix} c_1, n_1 \\ c_2, n_2 \\ c_3, n_2 \end{pmatrix}$ and $\begin{pmatrix} c_1, n_2 \\ c_2, n_2 \\ c_3, n_2 \end{pmatrix}$. If $n_1$ is always tested first when trying to place a component, starting from the component order $\prec c_1$, $c_2$, $c_3 \succ$, the search must fail to place $c_3$ as depicted in Figure 4.8.

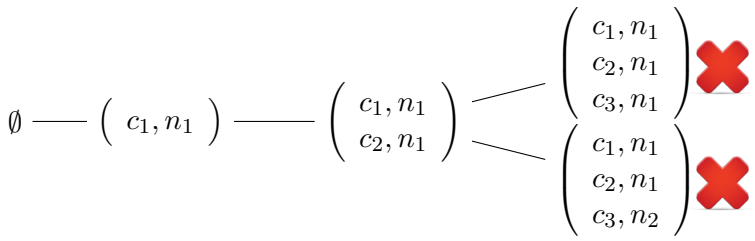


Figure 4.8: Heuristic Search Example—Search Process Under the Initial Component Order.

When failing to place a component *comp* (*e.g.,* $c_3$ in Figure 4.8), the heuristic DCO tries to move *comp* forward to test new component orders. Because of failures similar to $c_3$ in Figure 4.8, the following component orders are tested one by one:

$\prec c_1, c_2, c_3 \succ$ (fail to place $c_3$ because of $c_2$ placed in $n_1$)
$\prec c_3, c_1, c_2 \succ$ (fail to place $c_2$ because of $c_3$ placed in $n_1$)
$\prec c_2, c_3, c_1 \succ$ (fail to place $c_3$ because of $c_2$ placed in $n_1$)
$\prec c_3, c_2, c_1 \succ$

By searching under the first two orders $\prec c_1, c_2, c_3 \succ$ and $\prec c_3, c_1, c_2 \succ, c_3$ and $c_2$ are found leading to failures and moved forward. Then, $\prec c_2, c_3, c_1 \succ$ and $\prec c_3, c_2, c_1 \succ$ are tested, which respectively reorder $c_2$ and $c_3$. The search process under the final order (*i.e.,* $\prec c_3, c_2, c_1 \succ$) is given in Figure 4.9, which shows that the algorithm still fails to place $c_2$ after placing $c_3$ in $n_1$ (as in the upper branch of Figure 4.9). However, no new component order can be produced by moving $c_2$ forward (*i.e.,* $\prec c_2, c_3, c_1 \succ$ is already tested). Consequently, the algorithm backtracks as FirstFit, and finally finds out a solution.

Regarding the example above, a heuristic search can contain three phases: (1) identifying and moving forward components leading to failures (*e.g.,* $c_2$ and $c_3$ in the example); (2) adjusting the order of components moved forward (in the worst case, all possible orders of these components

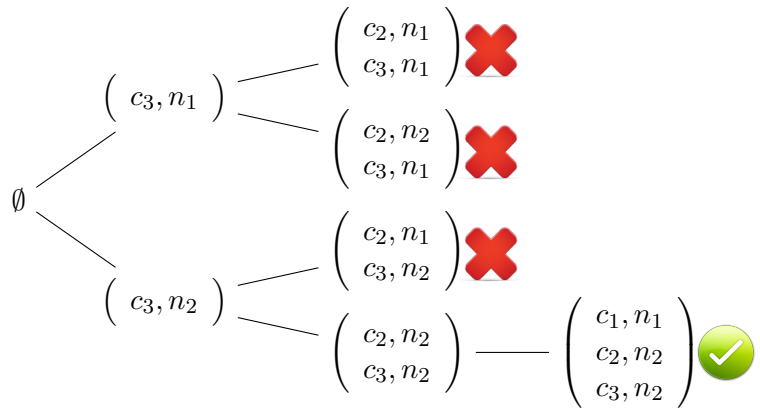


Figure 4.9: Heuristic Search Example—Search Process Under the Final Component Order.

are tested); (3) backtracking to deal with failures instead of changing components' order (in the worst case, the search space constructed by components moved forward is traversed, which is explained in the following).

Given a placement problem with $m$ fog nodes and $n$ components, if $k$ ($k \leq n$) components are moved forward during the heuristic search, the worst-case complexity of the heuristic algorithm can be expressed as follows:

$$\begin{aligned} &O(k \times n \times m && (1)\\ &+k! \times k \times m && (2)\\ &+m^k) && (3) \end{aligned}$$

Parts (1), (2), and (3) in this expression respectively correspond to phases (1), (2), and (3) of a heuristic search. In phase (1), to identify a component that can lead to a failure, at most $n \times m$ tests are needed (in the worst case, the failure is found at the last / $n$th component, and each of the other component is successfully placed until the last / $m$th fog node tested). Knowing that there are $k$ components to move forward, phase (1)'s complexity can reach $O(k \times m \times n)$. In phase (2), to verify if a component order needs to be adjusted, there can be at most $k \times m$ tests. The $k$ components moved forward have $k!$ possible orders, hence phase (2) can take time $O(k! \times k \times m)$. In phase (3), the $k$ components moved forward must be the first $k$ components to place, and no other component can lead to a failure (otherwise, the value of $k$ changes). As a backtrack must take place at a component failed to place, the search can change at most these $k$ components' hosts by backtracking. Considering that there are $m^k$ possibilities to place $k$ components, phase (3)'s complexity is $O(m^k)$.

Given the same placement problem, FirstFit's complexity is $O(m^n)$. Although the heuristics have a even higher worst-case complexity when $k$'s value is close to $n$, the complexity can be dramatically lowered when $k < n$ (with a problem-dependent threshold). Moreover, the heuristics help to

avoid the worst case and to lower the value of $k$, as detailed in the following.

By taking care of fog nodes' order (*e.g.,* testing $n_2$ first to place $c_2$ and $c_3$ in the example), the heuristics AFNO and DAFNO can reduce the number of tests for placing a component. By avoiding meaningless tests, FailCap reduces the number of tests for identifying components leading to failures. When testing to place a component, the number of tests is reduced by AFNO and DAFNO if this test can be passed, and is reduced by FailCap if this test must fail. Therefore, through these heuristics, the number of tests for each component can be much lower than $m$. Although AFNO, DAFNO, and FailCap do not guarantee to avoid the worst case (hence, $m$ is still used in the complexity expression), their counterexamples can be rare. By initially ordering components according to their bandwidth requirements (*e.g.,* ordering $c_2$ and $c_3$ before $c_1$ initially), InitCO can reduce the number of component orders to test in phases (1) and (2). By reordering components dynamically, DCO helps to avoid phase (3) in a search. By testing local fog nodes priorly, AFNO and DAFNO also help to reduce the number of components to move forward (*i.e.,* to lower $k$), test fewer component orders in phase (2), and avoid phase (3).

As stated in Section 4.3.4, each proposed heuristic has its cost. DCO's overhead—the two additional phases (*i.e.,* phases (1) and (2)) are already considered in the complexity expression. Regarding heuristic search's complexity, AFNO, DAFNO (which calculate anchors without considering all constraints), and InitCO's cost can be omitted. Differently, making it possible to miss proper fog nodes when testing to place a component, FailCap risks of increasing the number of components to move forward (*i.e.,* value of $k$) and the number of component orders to test. Such a risk increases when FailCap's parameter $FailNB$ is assigned to a low value, which at the meanwhile helps to reduce the number of tests under component orders leading to failures. Similar to $FailNB$, when DCO's parameter *stepLen* has a low value, the algorithm can have to test more component orders. However, as discussed in Section 4.3.2, a lower *stepLen* helps to reuse hosts found under previous component orders. Consequently, a certain value of $FailNB$ (or *stepLen*) can speed up or slow down the search, which depends on the problem to deal with. Further discussion and evaluation of these parameters are given in Chapter 5.

To sum up, the proposed heuristics introduce many mechanisms to accelerate the search, which help to: i) lower the number of components to move forward (*i.e.,* to lower $k$), and thereby decrease the algorithm's worst-case complexity (possible to be lower than that of FirstFit); ii) guide the search to approach its best case (whose complexity is $O(n)$) as much as possible. Different from the random search of FirstFit, only specific placement problems (*i.e.,* counterexamples of the heuristics' functionality) can lead to the heuristic search's worst case. In particular, by applying proposed heuristics, it is very probable to avoid phase (3), which lowers the complexity to

$O(k \times m \times n \ + \ k! \times k \times m)$.

## 4.5 Summary

To deal with the initial placement problem, this chapter:

- introduces a model and an objective function (*i.e.,* minimizing WAL) to formulate the problem;
- proposes a naive algorithm—FirstFit;
- proposes five heuristics (AFNO, DAFNO, InitCO, DCO, and FailCap) based on FirstFit. As shown in Figure 4.10, each proposed heuristic has its specific functionality (*i.e.,* manipulating the order of fog nodes / components or the number of fog nodes to test before / during the search), and thus all the five heuristics can be combined with each other;
- analyzes and compares algorithm complexity of FirstFit and the heuristic algorithm combining all proposed heuristics.

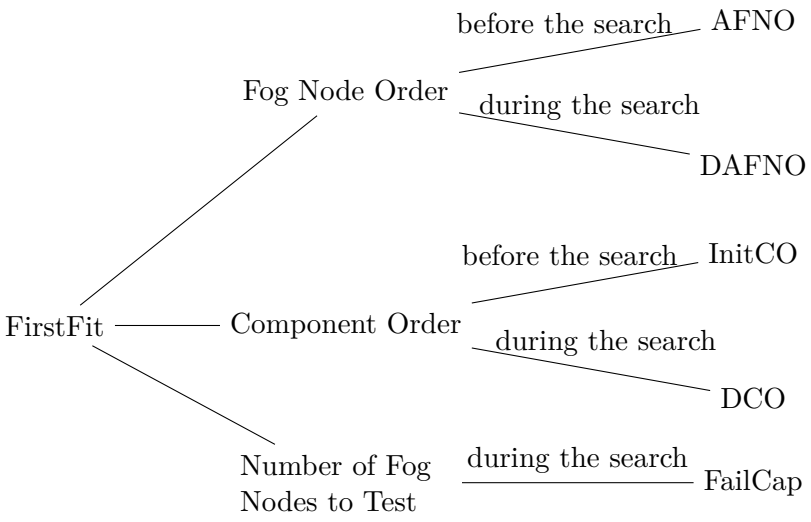


Figure 4.10: Heuristics Positioning.

An evaluation for validating the objective function and each heuristic proposed in this chapter is given in Chapter 5.

# 5 Evaluation of Proposed Initial Placement Approach

## Contents



This work deals with the problem of placing IoT applications in the fog. For solving the initial placement problem (*i.e.,* making initial placement decisions when no application is placed[1]), Chapter 4 proposes a model, an objective function (*i.e.,* minimizing Weighted Average Latency / WAL of placed applications), a backtrack algorithm (*i.e.,* FirstFit), and five heuristics (*i.e.,* AFNO, DAFNO, InitCO, DCO, and FailCap) that can be combined with each other.

This chapter evaluates Chapter 4's proposition with comparative simulation. The symbol "-" is used to indicate the combination of heuristics (*e.g.,* AFNO-InitCO-DCO(1)-FailCap($\infty$) means the combination of AFNO, InitCO, DCO with *stepLen* assigned to 1, and FailCap with $FailNB$ assigned to an infinite value). As stated in Section 4.3, there exists dependencies between proposed heuristics: DAFNO depends on AFNO; InitCO depends on AFNO / DAFNO; FailCap is based on both AFNO / DAFNO and DCO. For simplicity, DAFNO-AFNO is referred to as DAFNO in this chapter.

In the following, Section 5.1 details the evaluation environment and implementation. Two groups of evaluation based on two use cases are respectively given in Section 5.2 and Section 5.3. Finally, Section 5.4 concludes this chapter.

[1]The dynamic placement problem (for adjusting the placement after a placement decision is made) is discussed in Part II.

## 5.1 Evaluation Environment and Implementation

To compare different placement algorithms, each algorithm is evaluated from two aspects: scalability and result quality.

The scalability is assessed through: i) each algorithm's execution time (*i.e.,* processing duration) dealing with a same placement problem and ii) the maximal problem size (in terms of fog nodes' number and components' number) that each algorithm can deal with within a timeout. Given a placement problem, if an algorithm to evaluate has a definite initial component order (*i.e.,* being combined with InitCO), its execution time is measured three times to reduce the influence of the environment noise; otherwise, the algorithm is tested ten times for considering variances brought by random initial component orders. Execution times discussed in this chapter are average values of each group of measurements. The environment in which execution times are measured is detailed in Table 5.1.

| | |
|---|---|
| CPU | Intel i7 - 7820HQ @ 2.90 GHz |
| RAM | 16GB |
| OS | Debian 8.8 |
| JAVA | 1.8.0_131 |

Table 5.1: Execution Time Test Environment.

The result quality is in terms of considered applications' average response time. Based on SimGrid [40] simulation platform, applications' behaviors are simulated under placements to evaluate, which allows obtaining simulated response times and comparing placement decisions made by different algorithms. A response time discussed in this chapter is the average value of simulated response times of all placed applications.

As given in Figure 5.1, the implementation of the evaluation contains three modules: *Problem Generator*, *Placer*, and *Simulator*. *Problem Generator* is for generating infrastructure and application models used to evaluate proposed algorithms. Based on scripts, *Problem Generator* generates models of various applications and models of large-scale heterogeneous infrastructures. Each generated placement problem is composed of two files: $infra.xml$ for describing the infrastructure and *apps.xml* for describing applications to place, which are inputs of *Placer*. *Placer* corresponds to placement algorithms implemented in Java, which makes placement decisions. According to the placement decision, *Placer* outputs: i) $platform.xml$ for describing the infrastructure and *deploy.xml* for indicating each component's host, which are inputs of *Simulator*; ii) execution time of the evaluated algorithm. *Simulator* is responsible for the simulation. Based on interfaces provided by SimGrid, *Simulator* describes each component's calculation and communication behaviors (defined in use cases detailed in Section 5.2 and Section 5.3), and outputs simulated response

times of placed applications.

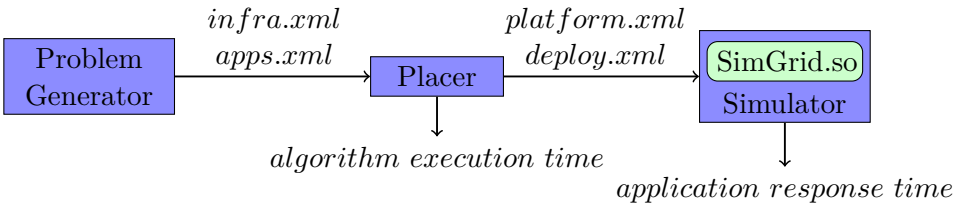


Figure 5.1: Implementation Schema.

## 5.2 Use Case 1: Smart Bell

### 5.2.1 Use Case Description

This use case concerns the placement of an IoT application "Smart Bell" [12]. Based on computer vision and household cameras, Smart Bell analyzes camera captured images and helps to control smart homes' door security. The cloud can not satisfy such an application because of:

- *high network latency.* The cloud is far from the cameras in terms of network latency. Consequently, transferring camera captured images to the cloud can be time-consuming.
- *high bandwidth consumption.* Transferring high-volume raw data (*i.e.,* camera captured images / video streams) to the cloud challenges the core network's bandwidth capacity.
- *privacy risk.* Transferring personal information (*e.g.,* camera captured images) to the cloud must pass by many networks, which raises the risk of violating the data privacy.

Benefiting from the proximity to end devices, the fog overcomes cloud's drawbacks and suits well Smart Bell. This use case's infrastructure and application are detailed in the following.

#### Fog Infrastructure

The infrastructure of this use case contains: clouds, Points of Presence (PoP[2]), and household devices. In each home, devices (*e.g.,* mobiles, PCs, cameras, and screens) are connected via a wireless gateway (*i.e.,* box). Two kinds of appliances, cameras and screens, respectively provide image capturing services and displaying services. An example of the infrastructure is given in Figure 5.2.

[2]Provided by telecom operators, a PoP designates a set of devices routing data flows between networks. For simplification, a PoP is considered as a single device in this work.

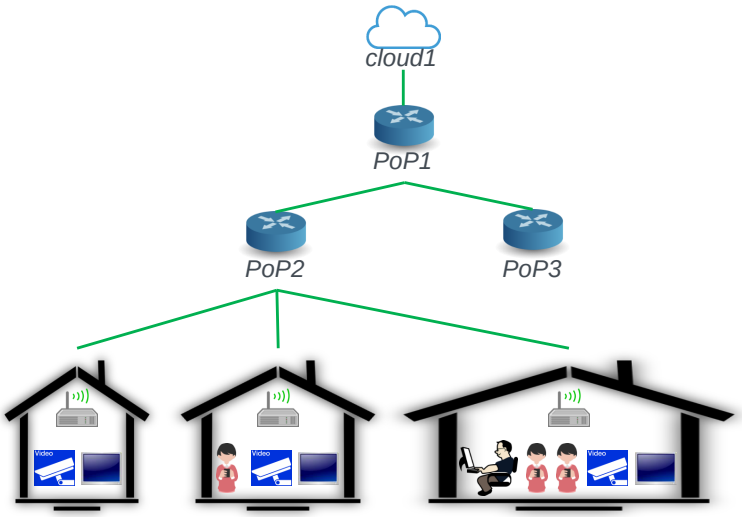


Figure 5.2: Infrastructure Example (each home is equipped with a camera, a screen, and a wireless gateway. *home*2 also has a mobile. *home*3 has two mobiles and a PC).

### Smart Bell Application

As a smart home (and smart neighborhood) application, Smart Bell notifies home inhabitants when they get a visitor. Each home served by Smart Bell has a camera to capture visitor images, a screen to display notifications, and a database (DB) to store images of the home inhabitants and their friends. The DB allows Smart Bell to identify the relationship between the inhabitants and each visitor and to make notifications correspondingly.

Consider a certain home, when a visitor rings at the door, Smart Bell verifies if he / she is an inhabitant or a friend according to the visited home's DB; if not, through communicating with neighbors' DBs, Smart Bell checks if the visitor is a neighbor's friend; if a visitor is not recognized, he / she is considered as a stranger, whose information (*i.e.,* captured images) is stored by Smart Bell for security reasons. If a stranger keeps ringing doorbells in a neighborhood, Smart Bell activates an alarm to ensure the security.

As listed in Table 5.2, Smart Bell has several component types. Each component type is in charge of one kind of analytical / storage tasks, and has a number of components to distribute these tasks. Because each home's DB stores private data, a DB must be placed in its home. Without special privacy requirement, other components can be placed anywhere.

Smart Bell can have multiple instances. Each instance of Smart Bell serves a neighborhood and functions independently of other instances. An example instance, which serves a neighborhood of three homes, is given in Figure 5.3. In Figure 5.3, each vertex in blue is a component, each vertex in green is an appliance, and each edge is a binding. Certain components

| Comp Type | Functionality |
|---|---|
| Extractor | extracting human faces in captured images |
| DB | storing inhabitants' and friends' information (e.g., face images) |
| Recognizer | trying to recognize visitors |
| Decider | making reaction decisions for each visitor |
| Executor | generating and sending commands to inform inhabitants through screens |
| Recorder | storing strangers' information and counting how many times a stranger appears |

Table 5.2: Component Types of Smart Bell.

are shared by multiple homes (*e.g., Extractor* − *a* is shared by *home*1 and *home*2, *Recognizer* is shared by the three homes).

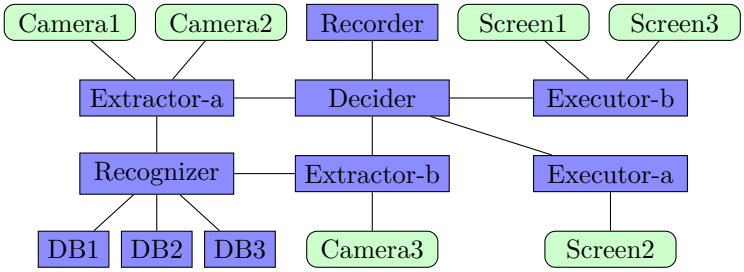


Figure 5.3: Example Instance of Smart Bell.

### 5.2.2 Evaluation Setup

As stated in Section 5.1, placement algorithms' inputs—infrastructure and application models are generated by scripts. This subsection details the model generation's attributes.

#### Fog Infrastructure

In the fog, each fog node provides certain CPU, RAM, and DISK resources. Generally, a fog node in a higher network position is more capable. For fog nodes in similar network positions, their capacities still can strongly differ, even for the ones of the same type (*e.g.,* two PCs). In order to consider such heterogeneity, each fog node's capacity randomly distributes in a range related to its type, as listed in Table 5.3.

Likewise, network link resources also follow uniform distributions. Network latency and available bandwidth ranges of each link type are listed in Table 5.4.

| **Device Type** | **CPU** (GFlops) | **RAM** (GB) | **DISK** (GB) |
|---|---|---|---|
| cloud | infinite | infinite | infinite |
| PoP | 0 ∼ 100 | 0 ∼ 500 | 0 ∼ 5000 |
| box | 0 ∼ 1 | 0 ∼ 1 | 0 ∼ 100 |
| PC | 0 ∼ 2 | 0 ∼ 4 | 0 ∼ 200 |
| mobile | 0 ∼ 1 | 0 ∼ 2 | 0 ∼ 50 |

Table 5.3: Capacity Ranges of Each Fog Node Type.

| **Link Type** | **LAT**(ms) | **BW**(MBps) |
|---|---|---|
| cloud – PoP | 30 ∼ 100 | 0 ∼ 1000 |
| PoP – PoP | 3 ∼ 7 | 0 ∼ 5000 |
| box – PoP | 1 ∼ 20 | 0 ∼ 100 |
| PC – box<br>mobile – box<br>camera – box<br>screen – box | 1 ∼ 2 | 0 ∼ 1000 |

Table 5.4: Capacity Ranges of Each Link Type.

### Smart Bell Application

From applications' point of view, each component relies on certain processing and storage resources. Resource requirements of each component type of Smart Bell are given in Table 5.5. As aforementioned, each DB must be placed in its home, while components of other types can be placed in any fog node, which results in different DZs (see Section 4.1 for Dedicated Zone).

| **Component Type** | **ReqCPU** (GFlops) | **ReqRAM** (GB) | **ReqDISK** (GB) | **DZ** |
|---|---|---|---|---|
| DB | 0.1 | 0.1 | 0.1 | Home |
| Recorder | 0.5 | 0.5 | 20 | Infra |
| Extractor | 0.2 | 0.2 | 0 | Infra |
| Recognizer | 0.3 | 0.3 | 0 | Infra |
| Decider | 0.2 | 0.1 | 0 | Infra |
| Executor | 0.1 | 0.2 | 0 | Infra |

Table 5.5: Requirements of Each Component Type.

Network resource requirements of each binding type of Smart Bell are listed in Table 5.6.

| **Binding Type** | **ReqLAT**(ms) | **ReqBW**(MBps) |
|---|---|---|
| Camera – Extractor | 25 | 0.6 |
| Screen – Executor | 25 | 0.01 |
| Extractor – Recognizer | 25 | 0.1 |
| DB – Recognizer | 25 | 0.3 |
| Decider – Recorder | 25 | 0.2 |
| Extractor – Decider | 50 | 0.1 |
| Decider – Executor | 50 | 0.01 |

Table 5.6: Requirements of Each Binding Type.

### 5.2.3 Result and Discussion

To evaluate proposed heuristics, three groups of evaluations are carried out. First, the proposed placement approach (*i.e.,* combined heuristics) is compared with other placement algorithms (*i.e.,* exact algorithm and metaheuristic). Then, different heuristic combinations are compared. Finally, parameter of proposed heuristics are discussed.

#### Heuristics vs Exact Algorithm and Metaheuristic

This evaluation compares the proposed heuristic algorithm with an exact algorithm ILP and a metaheuristic Genetic Algorithm (GA).

ILP is based on Integer Linear Programming, and is implemented using IBM CPLEX [29]. Given a placement problem, ILP always returns the optimal solution (*i.e.,* the solution that best fits the optimization objective). However, its execution time increases dramatically with problem size (*i.e.,* fog nodes' number and components' number).

Genetic Algorithm (GA) is a representative metaheuristic, which refines a population (*i.e.,* a set of placements). GA continuously generates new placements and adds them into the population. A generated placement inherits from the population (*i.e.,* for generating a new placement, the decision of a component's host is same as a placement in the population). Then, placements that worse fit the optimization objective are filtered out from the population, which helps to generate better placements in the next generation. Thus, GA is expected to outperform the naive approach—FirstFit (the pseudo code and more details of GA are given in Section 7.2.1).

A heuristic algorithm—DAFNO-InitCO-DCO(0.3) (which appears as the best heuristic combination according to evaluation results, see Section 5.4) is selected to be compared with ILP, GA, and FirstFit. Because of the high execution time (which can be days, weeks or even longer) of ILP / GA / FirstFit, we are not able to get their results for large-scale problems. To avoid the execution time explosion, this evaluation uses a single Smart Bell instance—the example in Figure 5.3 to place. The infrastructure

is similar to Figure 5.2, but it has 40 more homes connected to $PoP_2$ and $PoP_3$. Each added home has a box and a PC / mobile. To cover different resource situations, 10 infrastructure models are generated following resource capacity distributions defined in Section 5.2.2, which implies 10 placement problems.

Taking the 10 placement problems into account, each evaluated algorithm's average execution time and average response time are summarized in Table 5.7. For each problem, obtained response times are normalized according to ILP's, *i.e.,* the response time under the optimal placement is regarded as 1 (100%).

| **Algorithm** | **Execution Time** (s) | **Response Time** |
|---|---|---|
| FirstFit | 265 | 186.8% |
| ILP | 343 | 100% |
| GA[3] | 28 | 143.9% |
| DAFNO-InitCO-DCO(0.3) | 0.003 | 100.3% |

Table 5.7: Evaluation Results of Different Placement Algorithms.

Because of random component and fog node orders, FirstFit gets a high execution time and a high response time. Guaranteeing to return optimal solutions, ILP highly lowers the response time. However, it has an even higher execution time than FirstFit. GA outperforms FirstFit on both execution time and response time. Nevertheless, the response time obtained by GA is about 43% higher than ILP's optimal solutions. DAFNO-InitCO-DCO(0.3) improves FirstFit / GA on both execution time and response time. Compared with ILP, DAFNO-InitCO-DCO(0.3) gets a similar response time while introducing a 100000 times' speed-up.

Figure 5.4 depicts obtained response times versus WAL values. It can be found that a placement with lower WAL leads to a lower response time. The correlation between WAL value and Smart Bell's response time is 0.853, which validates the proposed objective function—minimizing WAL.

[3]GA's result given in Table 5.7 is based on the following parameter values: population size = 20, mutation rate = 0.01. See Section 7.2.1 for more details.

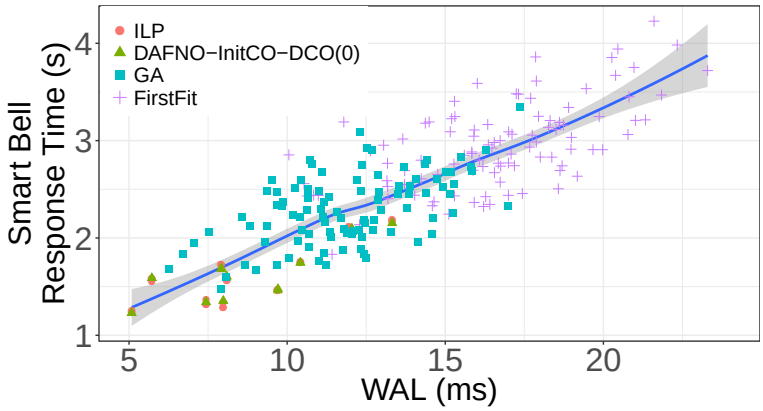


Figure 5.4: Smart Bell's Response Time under Placements with Different WAL Values[4].

### Heuristic Combinations' Comparison

According to the previous evaluation, ILP arrives to place 1 application (with 10 components) in an infrastructure (with 91 fog nodes) in 343 seconds. To compare different heuristic combinations' scalability, given the same execution time—343 seconds, we evaluate how many components each algorithm can place in a larger infrastructure containing: 1 cloud, 4 high-level PoPs (as PoP1 in Figure 5.2), 20 low-level PoPs (as PoP2 in Figure 5.2), and 5000 homes. The PoPs form a random connected graph, and each high-level PoP is connected with the cloud. Each home has a box, a camera, a screen and 0∼3 PCs and mobiles. Following resource capacity distributions given in Section 5.3.2, an infrastructure model is generated, which must cover different resource situations thanks to its large scale. 776 Smart Bell instances are generated to serve the 5000 homes. Each 3∼10 homes are in the same neighborhood that is served by a Smart Bell instance. In each Smart Bell instance, DB number equals to the number of homes served by the instance, component numbers of other types randomly distribute in 1∼3. Starting from 1 instance, each evaluated algorithm places more and more instances until the timeout is exceeded. Execution times of evaluated algorithms are depicted in Figure 5.5.

Due to the large infrastructure scale, FirstFit and GA[5] timeouts at the very beginning (*i.e.,* when dealing with a single instance). AFNO, DAFNO,

[4]As stated in Section 5.1, for each placement problem: FirstFit (or GA) is launched 10 times because of its randomness, which produces 10 different placements / points in Figure 5.4; DAFNO-InitCO-DCO(0) (or ILP) is launched 3 times, which get a same placement / point in Figure 5.4.

[5]The space complexity of ILP exponentially increases with the placement problem size. As a result, RAM needed by ILP highly exceeds the capacity of the test environment (*i.e.,* 16GB, see Table 5.1), and we are not able to get results of ILP in this evaluation.

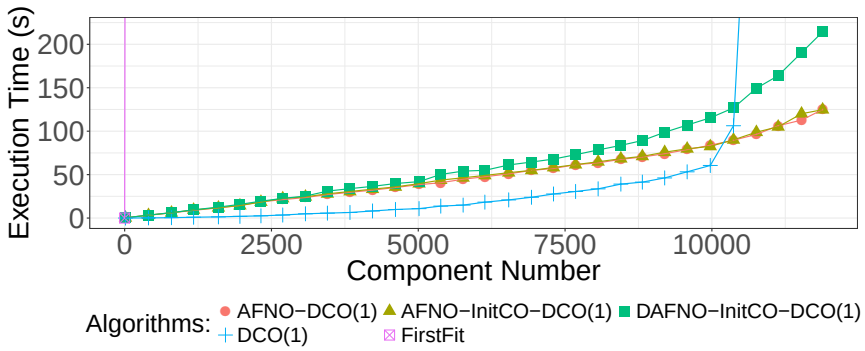


Figure 5.5: Execution Times with Growing Applications.

AFNO-InitCO, and DAFNO-InitCO only arrive to place 1 instance within the timeout.

As shown in Figure 5.5, by applying the heuristic DCO, DCO(1)[6] timeouts until placing more than 10500 components, which highly improves First-Fit on scalability. The heuristic DCO reorders components during the search for deriving to direct path to a solution, which helps to accelerate the search. However, after reordering components, DCO has to redo the search for certain components. This overhead becomes significant when dealing with more than 10000 components (DCO has to redo the search for more components after each reordering) and testing fog nodes in random orders (which results in more reorderings).

The heuristic AFNO calculates an anchor for each component. When trying to place a component, AFNO tests priorly fog nodes close to this component's anchor. According to Figure 5.5, by combining AFNO and DCO(1), AFNO-DCO(1) arrives to place all generated Smart Bell instances within the timeout, which further improves the algorithm's scalability. Because of the anchor-calculation, AFNO-DCO(1)'s execution time is higher than DCO(1) when the number of components is lower than 10000. However, when dealing with more than 10000 components, AFNO highly lowers the number of component reorderings performed by DCO, and thus accelerates the search.

AFNO-InitCO-DCO(1) and AFNO-DCO(1) get almost same execution times in this evaluation, which indicates that the overhead of the heuristic InitCO (*i.e.,* ordering components according to their bandwidth requirements, see Section 4.3.2) is insignificant.

The heuristic DAFNO is proposed to keep anchors up-to-date during the search. DAFNO-InitCO-DCO(1) also successfully places all Smart Bell instances under the timeout. However, because of the overhead of updating anchors, DAFNO-InitCO-DCO(1)'s execution time is always higher than

[6]In this evaluation, heuristic DCO's parameter *stepLen* is always assigned to 1. *stepLen* value is evaluated and discussed in the next evaluation.

AFNO-InitCO-DCO(1).

For each number of components, obtained response times are normalized according to DAFNO-InitCO-DCO(1)'s, *i.e.,* DAFNO-InitCO-DCO(1)'s response time is regarded as 1. The heuristic DCO only helps to accelerate the search without taking care of the result quality. As a result, DCO(1)'s response time is always higher than 200%. Response times of the other heuristic combinations are compared in Figure 5.6.

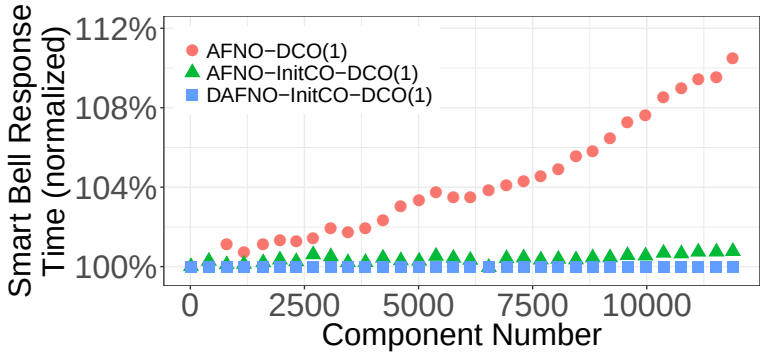


Figure 5.6: Response Times with Growing Applications.

As shown in Figure 5.6, AFNO-InitCO-DCO(1) has a lower response time than AFNO-DCO(1), and the difference increases with the number of components. Thus, we conclude that the heuristic InitCO, which prioritizes components with higher bandwidth requirements, helps to lower applications' response time, especially when placing a lot of components. DAFNO-InitCO-DCO(1)'s response time is always lower than AFNO-InitCO-DCO(1), which validates that dynamic anchors further lower applications' response time compared with static ones.

According to obtained execution times and response times, each of the heuristics evaluated in this evaluation (*i.e.,* DCO, AFNO, DAFNO, and InitCO) improves the algorithm's scalability and / or result quality.

### Impacts of Parameters *stepLen* and $failNB$

According to the previous evaluation, DAFNO-InitCO-DCO(1) (equivalent to DAFNO-InitCO-DCO(1)-FailCap($\infty$)) arrives to place 11886 components in an infrastructure (with 12483 fog nodes and 10000 appliances) in about 214 seconds. AFNO-InitCO-DCO(1) (equivalent to AFNO-InitCO-DCO(1)-FailCap($\infty$)) takes 124 seconds to solve the problem, and gets a response time that is 0.6% higher than DAFNO-InitCO-DCO(1). This placement problem is reused to evaluate DAFNO-InitCO-DCO(*stepLen*)-FailCap($failNB$) and AFNO-InitCO-DCO(*stepLen*)-FailCap($failNB$) with different parameter settings. The evaluation results of these two algorithms are respectively depicted in Figure 5.7 and Figure 5.8.

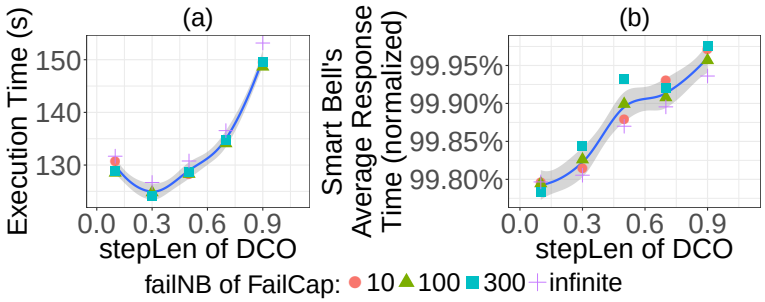


Figure 5.7: DAFNO-InitCO-DCO($stepLen$)-FailCap($failNB$)'s Results.

According to the principles of the heuristic DCO (see Section 4.3.2), a lower (*resp.*, higher) *stepLen* leads to more (*resp.*, fewer) component moves, but more (*resp.*, less) reutilization of obtained search results. According to Figure 5.7 (a), a proper *stepLen* value can further accelerate the search, and gets an execution time lower than 214 seconds. Considering that the execution time explodes (*i.e.*, higher than 30 minutes) when $stepLen = 0$, *stepLen* values near to the two extremes (*i.e.*, 0 and 1) can cause high execution times. In this evaluation, a $failNB$ value lower than $\infty$ leads to lower execution times. However, compared with the execution time lowered by *stepLen*, the difference brought by different $failNB$ values is insignificant.

A lower *stepLen* changes less strongly components' order, which better respects the initial component order produced by InitCO. According to response times (normalized according to DAFNO-InitCO-DCO(1)'s) shown in Figure 5.7 (b), lower *stepLen* values help to get lower response times. However, a low $failNB$ risks of missing proper fog nodes and resulting in a higher response time.

According to Figure 5.8, compared with DAFNO-InitCO-DCO($stepLen$)-FailCap($failNB$), AFNO-InitCO-DCO($stepLen$)-FailCap($failNB$) gets a lower execution time and a higher response time. Such a difference is because of the overhead and the functionality of the heuristic DAFNO, which updates anchors during the search. In these two algorithms, the parameter *stepLen* (*resp.*, $failNB$) has similar impacts, which are already discussed in the analysis of Figure 5.7.

This evaluation shows that a low *stepLen* value of DCO helps to further improve the heuristic combination's result quality. However, a *stepLen* close to 0 causes high execution times when dealing with large-scale problems. A $failNB$ value lower than $\infty$ decreases the execution time while raising the response time. Hence, a trade-off is needed for each parameter to consider both execution time and response time.

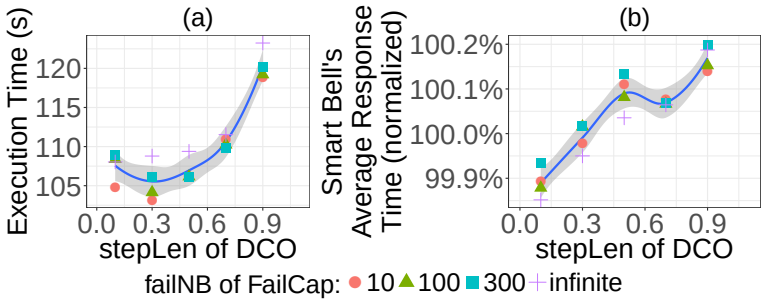


Figure 5.8: AFNO-InitCO-DCO($stepLen$)-FailCap($failNB$)'s Results.

## 5.3 Use Case 2: Data Stream Processing

To evaluate placement algorithms with various applications (rather than a specific one, as Smart Bell), applications are randomly generated in this use case.

### 5.3.1 Use Case Description

The continuous increase of end devices connected to the internet is leading to an explosion of sensor-generated data. Such data contains valuable information, but most of the data is valuable only when processed under a certain delay. Thus, a lot of Data Stream Processing (DSP) applications [5, 41] are time-sensitive. Fog computing, which provides local processing resources, is right an enabler of such DSP applications.

#### Fog Infrastructure

The infrastructure of this use case contains clouds, edge servers, gateways, end fog nodes, and appliances as devices. An example is given in Figure 5.9.

#### Data Stream Processing Application

A DSP application can be represented as a directed acyclic graph, which contains three kinds of vertexes: data source, operator, and data consumer. A *data source* continuously generates a data stream; an *operator* receives and processes incoming streams, and then produces outgoing streams transferred to following operators or data consumers; a *data consumer* only receives and processes incoming streams.

An example DSP application, which maintains a database that stores the information of traffic loads, is given in Figure 5.10. In this example, data sources—traffic sensors sense traffic loads, whose raw data are filtered and aggregated by operators—filters and aggregators, respectively. The data consumer—knowledge base updates stored information in real-time, which can be used as a base of smart traffic lights / traffic route planning.

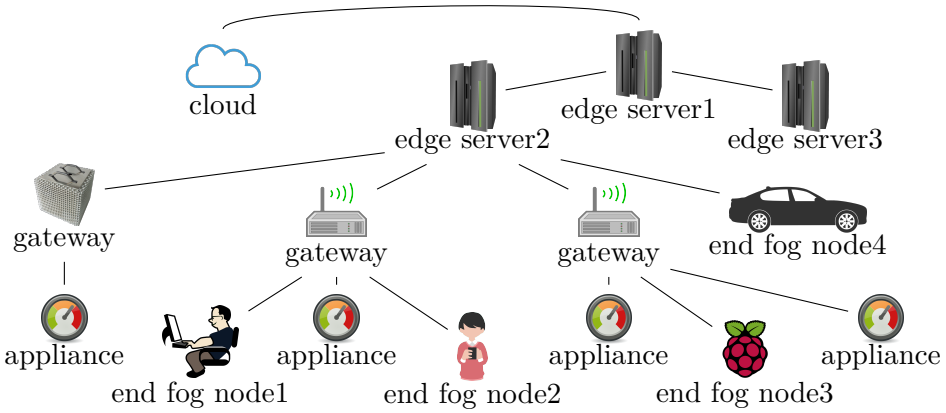


Figure 5.9: Infrastructure Example (containing a cloud, three edge servers, three gateways, four end fog nodes, and four appliances).

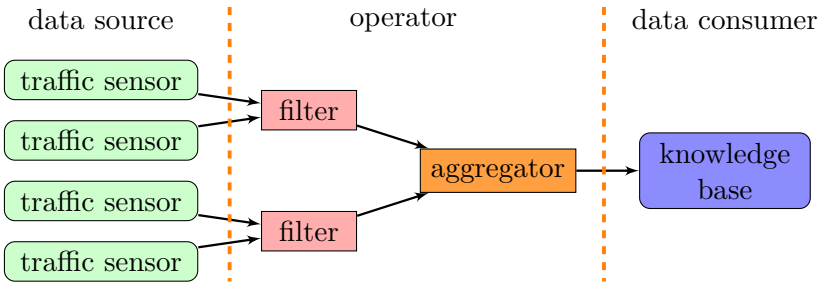


Figure 5.10: DSP Application Example—Traffic Knowledge Base (containing four road traffic sensors as data sources, three operators of two types, and one data consumer).

DSP applications are widely used to process data streams in many domains, such as smart transport / building / home, etc. In this use case, similar to the example in Figure 5.10, each DSP application has a set of data sources, operator types, and data consumers. Each operator type is in charge of one kind of analytical tasks (*i.e.,* to deal with a kind of incoming streams), and has a set of operators to distribute these tasks.

### 5.3.2 Evaluation Setup

This subsection details attributes for generating infrastructure / application models in this use case.

#### Fog Infrastructure

For generating infrastructure models, resource ranges of each fog node type and that of each network link type are respectively listed in Table 5.8 and

Table 5.9.

| **Device Type** | **CPU** (GFlops) | **RAM** (GB) | **DISK** (GB) |
|---|---|---|---|
| cloud | infinite | infinite | infinite |
| edge server | 0 ∼ 100 | 0 ∼ 500 | 0 ∼ 5000 |
| gateway | 0 ∼ 8 | 0 ∼ 10 | 0 ∼ 500 |
| end fog node | 0 ∼ 2 | 0 ∼ 4 | 0 ∼ 200 |

Table 5.8: Capacity Ranges of Each Fog Node Type.

| **Link Type** | **LAT**(ms) | **BW**(MBps) |
|---|---|---|
| cloud – edge server | 30 ∼ 100 | 0 ∼ 1000 |
| edge server – edge server | 3 ∼ 10 | 0 ∼ 1000 |
| gateway – edge server | 1 ∼ 20 | 0 ∼ 100 |
| end fog node – edge server | 10 ∼ 25 | 0 ∼ 100 |
| end fog node – gateway<br>appliance – gateway | 1 ∼ 5 | 0 ∼ 1000 |

Table 5.9: Capacity Ranges of Each Link Type.

### Data Stream Processing Application

A generated DSP application contains 1∼10 data sources, 1∼5 operator types, and 1∼10 data consumers[7]. Each operator type has 1∼10 components as its operators and a probability of 10% to be assigned a specific DZ composed of randomly selected fog nodes. Each generated application's data sources, data consumers, and operators communicate with each other via bindings.

To ensure DSP applications' proper execution, each component type and binding type requires certain amount of resources, as given in Table 5.10.

| component type requirements | | |
|---|---|---|
| ReqCPU | 0.1∼1 | GFlops |
| ReqRAM | 0.1∼1 | GB |
| ReqDISK | 0 ∼2 | GB |

| binding type requirements | |
|---|---|
| ReqLAT | 25 ∼50 ms |
| ReqBW | 0.01∼ 1 MBps |

Table 5.10: Resource Requirements of DSP Applications.

As composition elements of data streams, *data units* are produced by data sources and operators. Each data unit is characterized with a size and a computational amount, which respectively indicate the data amount to transfer and needed processing effort.

[7]As this work focuses on IoT applications' placement, data sources / consumers are components or randomly selected devices, and at least one of the two are devices.

Given a data unit of a data source, its response time is the time spent from its sending until that a data consumer receives its response. To simulate DSP applications' response times, the following assumptions are made: i) all data sources generate data units at a frequency of 1 data unit/s[8]; ii) considering that higher $ReqBW$ is required by bindings transferring larger data units, and that higher $ReqCPU$ is required by components dealing with higher computational amounts, a data unit's size and computational amount are respectively related to their corresponding $ReqBW$ and $ReqCPU$[9].

### 5.3.3 Result and Discussion

Three groups of evaluations are given in the following, which are respectively for comparing: i) the proposed heuristic algorithm with other placement algorithms (*i.e.,* exact algorithm and metaheuristic); ii) different heuristic combinations; iii) different parameter settings.

#### Heuristics vs Exact Algorithm and Metaheuristic

This evaluation compares a heuristic algorithm—DAFNO-InitCO-DCO(0.3) with ILP, GA, and FirstFit (see Section 5.2.3 for more details about ILP and GA). To avoid execution time explosion, a small-scale infrastructure and a single application are used as the placement problem. Similar to Figure 5.9, the infrastructure contains 1 cloud, 3 edge servers, and 3 gateways connected to EdgeServer-1, but it has 20 end fog nodes randomly connected to the three gateways and EdgeServer-1, and 40 appliances randomly connected to the gateways. A random DSP application is generated to place, which has 10 sensors as data sources, 2 operator types with 8 and 5 operators respectively, and 3 components as data consumers. To cover different resource situations, 10 infrastructure models are generated following resource capacity distributions defined in Section 5.3.2, which implies 10 placement problems.

Taking the 10 placement problems into account, each evaluated algorithm's average execution time and average response time are summarized in Table 5.11. For each problem, obtained response times are normalized according to ILP's, *i.e.,* the response time under the optimal placement is regarded as 1 (100%).

[8]For data sources with higher (*resp.,* lower) frequency, their data units can be considered as aggregated (*resp.,* divided) to adapt to the frequency.

[9]A data unit's size distributes in $0.01 \times ReqBW \sim 0.1 \times ReqBW$, its computational amount distributes in $0.01 \times ReqCPU \sim 0.1 \times ReqCPU$.

| Algorithm | Execution Time (s) | Response Time |
|---|---|---|
| FirstFit | 756 | 230% |
| ILP | 258 | 100% |
| GA[10] | 32.17 | 179% |
| DAFNO-InitCO-DCO(0.3) | 0.016 | 109% |

Table 5.11: Evaluation Results of Different Placement Algorithms.

In this use case, FirstFit performs the worst, which gets the highest execution time and the highest response time. ILP gets the lowest response time. However, because of the guarantee of returning optimal solutions, ILP's execution time is relatively high. GA outperforms FirstFit on both execution time and response time. Nevertheless, the response time obtained by GA is about 79% higher than ILP. DAFNO-InitCO-DCO(0.3) improves FirstFit / GA on both execution time and response time. Compared with ILP, although DAFNO-InitCO-DCO(0.3) gets a 9% higher response time, it introduces a 10000 times' speed-up.

Figure 5.11 shows obtained response times versus WAL values. The response time increases with WAL, and their correlation is up to 0.969, which validates the proposed objective function—minimizing WAL.

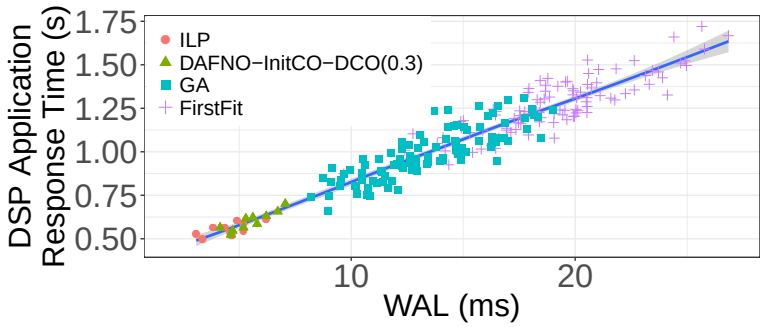


Figure 5.11: DSP Applications' Response Time under Placements with Different WAL Values.

### Heuristic Combinations' Comparison

According to the previous evaluation, ILP arrives to place 1 application (with 16 components) in an infrastructure (with 27 fog nodes and 40 appliances) in 258 seconds. To compare different heuristic combinations' scalability, given the same execution time—258 seconds, we evaluate how many applications / components each algorithm can place in a larger infrastructure containing: 1 cloud; 10 high edge servers (as edge server1 in Figure 5.9)

[10]GA's result given in Table 5.11 is based on the following parameter values: population size = 20, mutation rate = 0.01. See Section 7.2.1 for more details.

randomly connected to each other, and each of them also connects with the cloud; 50 low edge servers (as edge server2 in Figure 5.9), each connected with a random high edge server; 500 gateways, each connected with a random low edge server; 10000 end fog nodes, each connected with a random low edge server or gateway; 20000 appliances, each connected with a random gateway. Following resource capacity distributions given in Section 5.3.2, an infrastructure model is generated, which must cover different resource situations thanks to its large scale. Each algorithm places more and more applications until it exceeds the timeout of 258 seconds. Starting from 1 application, 25 randomly generated DSP applications are added each time. Execution times of evaluated algorithms are depicted in Figure 5.12.

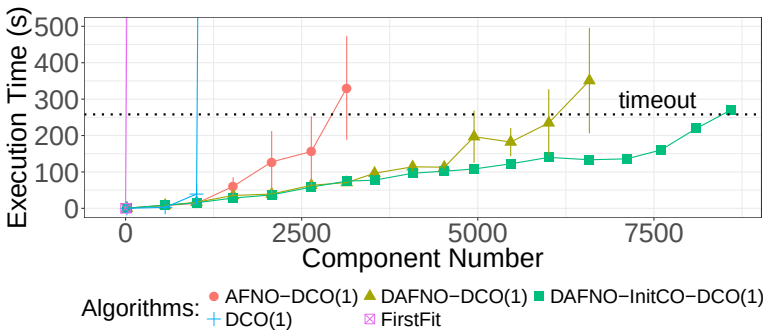


Figure 5.12: Execution Times with Growing Applications.

Among algorithms evaluated in Figure 5.12, an algorithm with more heuristics applied / combined is shown to be more scalable. Due to the large infrastructure scale, FirstFit and GA[11] timeout at the very beginning (*i.e.,* when dealing with a single application). AFNO, DAFNO, AFNO-InitCO, and DAFNO-InitCO only arrive to place 1 application within the timeout.

The heuristic DCO reorders components to avoid backtracks (which can cause high execution times) during a search. As shown in Figure 5.12, by applying DCO, DCO(1)[12] arrives to place 51 applications within the timeout. Compared with FirstFit, DCO(1)'s improvement validates that DCO helps to accelerate the search. However, after reordering components, DCO has to redo the search for certain components. This overhead is more significant when dealing with more components (DCO has to redo the search for more components after each reordering) and / or testing fog nodes in random orders (which results in more reorderings).

As shown in Figure 5.12, AFNO-DCO(1) timeouts until placing 151 applications, which further improves the algorithm's scalability. Because of

[11]Because RAM needed by ILP highly exceeds the capacity of the test environment (*i.e.,* 16GB, see Table 5.1), we do not arrive to get results of ILP.

[12]In this evaluation, heuristic DCO's parameter *stepLen* is always assigned to 1. *stepLen* value is evaluated and discussed in the next evaluation.

the anchor-calculation carried out by the heuristic AFNO, AFNO-DCO(1)'s execution time is higher than DCO(1) when the number of components is lower than 1000. However, by guiding the search to test local fog nodes priorly, AFNO lowers the number of component reorderings performed by DCO. When dealing with more than 1000 components, AFNO-DCO(1) out performs DCO(1), which validates that anchors help to accelerate the search.

Because anchors calculated by AFNO can be outdated during the search, the heuristic DAFNO is proposed to update anchors dynamically. When the number of components is lower than 1500, DAFNO-DCO(1) has a higher execution time than AFNO-DCO(1), which is caused by the overhead of updating anchors. However, with more applications concurrent to limited resources, anchors are easier to be outdated. Thus, DAFNO-DCO(1) outperforms AFNO-DCO(1) when dealing with more than 1500 components, which shows that up-to-date anchors help to further accelerate the search.

With random initial component orders, DCO(1), AFNO-DCO(1), and DAFNO-DCO(1) get higher variances than DAFNO-InitCO-DCO(1). According to Figure 5.12, DAFNO-InitCO-DCO(1) gets a better performance than DAFNO-DCO(1), especially when dealing with more than 5000 components. This is because of that, among considered constraints (see Section 4.1), the constraint of bandwidth consumption is the most complicated for the following reasons: i) a link's bandwidth capacity can be consumed by multiple bindings; ii) a binding can pass by multiple links; iii) where a binding is placed can concern two components' hosts; iv) one component can be connected by multiple bindings. Compared with other constraints (*e.g.,* CPU, RAM, DISK), DCO can have to test many more component orders to avoid a failure caused by a bandwidth capacity violation. Taking this complexity into account by satisfying priorly components with high bandwidth requirements (see Section 4.3.2), InitCO helps to reduce the number of components' reorderings carried out by DCO and accelerate the search.

For each application / component number, obtained response times are normalized according to DAFNO-InitCO-DCO(1)'s, *i.e.,* DAFNO-InitCO-DCO(1)'s response time is regarded as 1. Regarding that DCO(1)'s response time is always higher than 200%, response times of the other heuristic combinations are compared in Figure 5.13.

As shown in Figure 5.13, DAFNO-DCO(1)'s response time is always lower than AFNO-DCO(1), which validates that dynamic anchors further lower applications' response time compared with static ones. According to that DAFNO-InitCO-DCO(1) has a lower response time than DAFNO-DCO(1), and the difference increases with the number of components, we conclude that InitCO also helps to reduce applications' response time, especially when placing a lot of applications / components.

According to obtained execution times and response times, each of the heuristics evaluated in this evaluation (*i.e.,* DCO, AFNO, DAFNO, and InitCO) improves the algorithm's scalability and / or result quality.

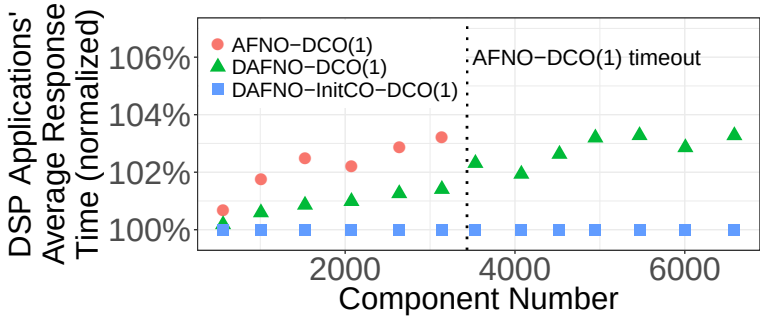


Figure 5.13: Response Times with Growing Applications.

### Impacts of Parameters *stepLen* and $failNB$

According to the previous evaluation, DAFNO-InitCO-DCO(1) (equivalent to DAFNO-InitCO-DCO(1)-FailCap($\infty$)) arrives to place 426 applications (with 8589 components) in an infrastructure (with 10561 fog nodes and 20000 appliances) in about 270 seconds. This placement problem is reused to evaluate DAFNO-InitCO-DCO(*stepLen*)-FailCap($failNB$) with different parameter settings. The evaluation results are depicted in Figure 5.14.

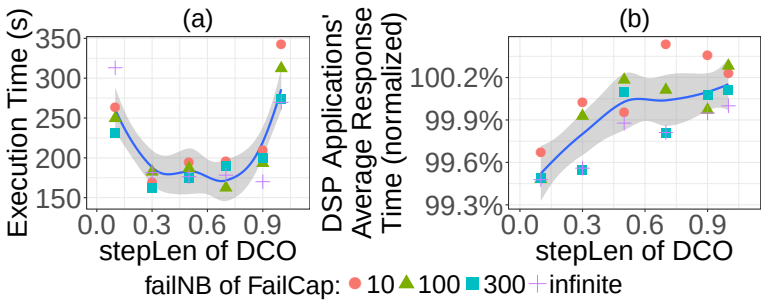


Figure 5.14: Evaluation Results with Different Parameters[13].

As stated in Section 4.3.2, a lower (*resp.,* higher) *stepLen* leads to more (*resp.,* fewer) component moves, but more (*resp.,* less) reutilization of obtained search results. According to execution times shown in Figure 5.14 (a), *stepLen* values near to the two extremes (*i.e.,* 0 and 1) can cause high execution times. Compared with DCO(1), a proper *stepLen* can further accelerate the search. On the other hand, compared with $\infty$, a lower $failNB$ helps to lower execution times only when *stepLen* is assigned to a low value (*e.g.,* as in Figure 5.14 (a) when $stepLen = 0.1$). The reduction of execution time becomes weak when $stepLen \geq 0.3$. Moreover, a low $failNB$ can miss proper fog nodes, cause more component reorderings, and get even higher execution times (*e.g.,* as in Figure 5.14 (a) when $stepLen = 1$).

A lower *stepLen* changes less strongly components' order, which bet-

[13]Results of DAFNO-InitCO-DCO(0)-FailCap($failNB$) are not obtained because of execution time explosion.

ter respects the initial component order produced by InitCO. According to response times (normalized according to DAFNO-InitCO-DCO(1)'s) shown in Figure 5.14 (b), lower *stepLen* values help to get lower response times. However, a low $failNB$ risks of missing proper fog nodes and resulting in a higher response time.

This evaluation shows that a low *stepLen* value of DCO helps to further improve the heuristic combination's result quality. However, a *stepLen* close to 0 causes high execution times when dealing with large-scale problems. Hence, a trade-off is needed to consider these two aspects. On the other hand, when *stepLen* is assigned to a proper value that helps to avoid high execution times, a $failNB$ value lower than $\infty$ no longer reduces or even raises execution times.

## 5.4 Conclusion

This chapter evaluates the proposed initial placement approach with two use cases. Use case 1 uses a specific application Smart Bell (see Section 5.2), in which each Dedicated Zone is composed of fog nodes close to each other (*i.e.,* fog nodes in a home). Use case 2 (*i.e.,* Data Stream Processing, see Section 5.3) uses randomly generated applications, a Dedicated Zone is composed of random fog nodes, which can be far from each other. Moreover, in use case 1, each home is served by a single Smart Bell instance. Consequently, resources in use case 2 can be more constrained compared with use case 1. Because of these differences, the heuristic FailCap performs better in use case 1, and heuristics DAFNO and InitCO perform better in use case 2.

Taking evaluation results obtained in these two use cases into account, DAFNO-InitCO-DCO(*stepLen*) appears as the best compromise in terms of scalability and result quality (based on our experience with the evaluated use cases, a *stepLen* value of 0.2∼0.4 is recommended). It gets solutions close to optimal ones obtained by ILP with much lowered execution times. It highly improves FirstFit / GA on both scalability and result quality. This algorithm also outperforms other heuristic combinations in most of the evaluated placement problems. DAFNO-InitCO-DCO(*stepLen*) is highly scalable, through which we get a satisfactory placement of more than 8000 components in an infrastructure with more than 10000 fog nodes within 200 seconds. Moreover, being able to deal with highly random problems (with random infrastructures / applications in terms of resource capacities / requirements and topologies, as discussed in Section 5.2.2 and Section 5.3.2), the proposition is shown to be a generic approach.

# Part II

# Dynamic Placement

# 6 State of the Art of Dynamic Placement

Contents



This chapter presents dynamic placement's state of the art. Section 6.1 gives the problem description, and introduces several criteria that must be considered by dynamic placement approaches in the context of fog. Section 6.2 discusses related works.

## 6.1 Problem Description and Criteria

In the fog, because of constantly varying applications and volatile end devices, there are several types of dynamicity:

- *applications' arrival / departure.* Requests for deploying new applications (and for removing deployed applications) arrive from time to time, which make applications to place dynamic.
- *devices' mobility.* Being possible to move to different places, a portable device can change its access point to the internet, which makes the infrastructure's network topology dynamic.
- *devices' churn.* An end device can leave (*e.g.,* lose network connection, be turned off) or join the fog at any time, which makes devices of the infrastructure dynamic.

Caused by these dynamic changes, applications' placement needs to be continuously adjusted to keep placed applications being executed properly and to fit the optimization objective. Without adjusting the placement (*i.e.,* changing certain components' hosts), an initial placement approach (*e.g.,* the approach proposed in Chapter 4) can not suit dynamic environments. To deal with such problems, dynamic placement approaches are designed. In particular, for addressing dynamic placement problems in the context of fog and IoT, several criteria must be considered, as detailed in the following.

**Dynamicity Type Coverage**. As the fog introduces different types of dynamicity (*i.e.,* applications' arrival / departure, devices' mobility, and devices' churn), a dynamic placement approach in the context of fog should be able to deal with all these dynamicity types.

**Migration Cost Awareness**. If a dynamic placement decision places a component in a device other than its current host, this component needs to be migrated. A component's migration is resource-consuming and has a negative influence on deployed applications' performance. Thus, a dynamic placement decision-making approach must consider the migration cost. If an approach does not take the migration cost into account (*i.e.,* the cost of migrating a component = 0), it can result in a lot of resource-consuming migrations and even cause a network congestion. If an approach does not allow migrating any component (*i.e.,* the cost of migrating a component = $\infty$), selected hosts' quality can get worse and worse, because each component's host is selected only once before unpredictable dynamic changes take place. Hence, in a highly dynamic fog infrastructure, placement approaches should be aware of components' migration cost to keep the placement close to the optimum.

**Reactivity**. For ensuring applications' proper execution (*i.e.,* ensuring that all constraints for placing considered applications are respected), each application's requirements must be respected. Namely, a placement decision must conform to several kinds of constraints (see Section 4.1 for more details). However, a dynamic change can violate certain constraints and makes the current placement invalid. In this case, an adjustment of the placement, which repairs the placement and allows applications being executed properly again, must be performed as soon as possible. Thus, a dynamic placement approach must make decisions rapidly to be reactive to dynamic changes.

**Application Agnostic**. As the fog is not dedicated to a single application or a specific type of applications. A placement approach should be generic enough to deal with any application.

**Location Dependency**. Based on sensing / actuating services, an IoT application is tied to its sensors / actuators, whose locations must be considered when making placement decisions.

To deal with the dynamicity in the context of fog and IoT, a dynamic placement approach must cover aforementioned dynamicity types, be aware of migration-cost, make placement decisions in reactive to dynamic changes, be application-agnostic, and support location-dependent applications.

## 6.2 Related Works of Dynamic Placement

For studying related works, criteria stated in Section 6.1 are used to evaluate them, as detailed in the following.

[42] deals with the placement problem in an infrastructure composed of edge devices. In this approach, each considered application serves one end user, and is tied to the user's mobile phone. Considering end users' mobility, an application's placement must be adjusted according to its end user's (or his / her mobile's) location. This approach assumes that the infrastructure is

always resource-rich (*i.e.,* any device / link in the infrastructure has enough resources to host all considered applications). For placing applications close to their end users, [42] proposes three strategies:

- *always migrate* strategy, which always migrates an application *app* to the nearest device to *app*'s end user.
- *infrequently migrate* strategy. According to this strategy, after migrating an application *app*, *app* can not be migrated again in a predefined time interval.
- *moving average* strategy. In *always migrate* and *infrequently migrate* strategies, each application's host is selected according to devices' current network latencies to the application's end user (or his / her mobile). Instead of the current network latency, *moving average* strategy selects the device with the lowest moving average network latency[1] to an application's end user as this application's host.

To avoid too many migrations, which can be incurred by *always migrate* strategy, *infrequently migrate* strategy introduces a "non-migration" period for each migrated application. For the same sake, based on *moving average* strategy, a device is selected as an application's host only if it has a lower average network latency during a period, which helps to make an application's host more stable. This approach deals with the dynamicity of mobiles' mobility, and supports applications that depend on users' locations. The proposed strategies only calculate each application's nearest device without introducing much calculation, and thus can be reactive to dynamic changes. However, none of proposed strategy models the migration cost. Furthermore, the assumption of the resource-rich infrastructure is not realistic. Given an infrastructure with limited resources, placements returned by this approach, which does not consider any constraint, can be invalid. Based on an assumption that each application serves one and only one end user, this approach is not application-agnostic.

[43] places applications tied to end users' mobiles in an infrastructure composed of cloud DCs and edge devices. It is assumed that:

- each mobile continuously notifies the infrastructure with the end user's location;
- when an end user moves, and the nearest (*i.e.,* in terms of network latency or hop number) edge device to him / her changes to $d$, a request for deploying his / her application is sent to $d$.

Considering the limit of available resources in the infrastructure, [43] proposes two strategies for deciding which application to satisfy first when a device receives multiple deployment requests:

[1] A moving average network latency is calculated by weighting and summing the current latency and a number of latencies measured in the past. The sum of all weights equals to 1. The longer (in terms of time) a latency is measured from now, the lower its weight is.

- *first come first serve* strategy. As named, based on this strategy, an application that arrives earlier is satisfied first.
- *delay-priority* strategy, which prioritizes applications that are more time-sensitive (*i.e.,* with stricter constraint of network latency).

A placement decision made by this approach always satisfies constraints of applications' resource requirements. However, based on *first come first serve* strategy, a later arrived application can be constrained by former arrived ones. Without considering the possibility of migrating former placed applications, placement decisions made by *first come first serve* strategy can get further and further from the global optimum along with dynamic changes. In contrast, based on delay-priority strategy, an application is migrated as long as it takes the resource needed by an application requiring lower latency. Such a decision is made without taking the migration cost into account, which can result in a lot of migrations and even a network congestion. The proposed strategies deal with the dynamicity of applications' arrival / departure, and support location-dependent applications. These strategies only order applications without introducing much calculation, and thus can be reactive to dynamic changes. However, without considering either the possibility of migrating components or components' migration cost, these strategies are not aware of migration cost. Because of the assumption that each application serves one and only one end user, this approach is not application-agnostic.

[44] places applications in a cloud DC composed of distributed devices. Aiming at balancing network resource utilization in the DC, a heuristic algorithm is proposed to deal with applications' arrival / departure. Upon an application's departure, this algorithm lowers the most congested link's load by migrating a subset of components consuming this link's resource. Based on a probabilistic model, the probability of migrating a component decreases with the number of migrated components in one placement adjustment, which helps to avoid network congestion caused by migrations. However, upon an application's arrival, this algorithm simply places the application in devices connected by underloaded links. Without considering components' migration when an application arrives, this algorithm takes only part of the search space into account, and can not find the optimum if it needs to migrate certain components. The proposed heuristic guides the search to a satisfactory result, which accelerates the search to be reactive to dynamic changes. Without any assumption on considered applications, this approach is application-agnostic. However, this approach only discusses the dynamicity of applications' arrival / departure. Being in the context of cloud, considered applications are not location-dependent. Because components' migration is not allowed upon applications' arrival, the migration cost awareness is considered to be partially supported.

[45] aims at maximizing the financial revenue of the infrastructure

provider. To address the dynamic placement problem, a lazy algorithm is proposed, which adjusts the placement only when it can be significantly improved. Based on this algorithm, a placement adjustment is applied only if it introduces an improvement (*i.e.,* financial cost reduction) higher than a predefined threshold. Such an algorithm helps to avoid:

- frequently migrating components;
- continuously migrating a component between two devices (*i.e.,* ping-pong effect).

However, when a dynamic change takes place, this lazy algorithm can cause a period waiting for more dynamic changes (until an adjustment introduces a significant improvement). Consequently, this approach can be not reactive enough to dynamic changes. Moreover, after such a waiting period, there can be a lot of components to migrate at one adjustment, which risks of causing a network congestion. To fit the optimization objective, the cost of components' migration is modeled in this approach. Without any assumption on considered applications, [45] is application-agnostic. However, being in the context of cloud, the considered applications are without location dependency, and only the dynamicity of applications' arrival / departure is dealt with.

[46] ensures applications' performance by dynamically reconfiguring deployed applications. If an application is overloaded, new instances (or components) of this application are deployed to lower other instances' charge. For underloaded applications, deployed instances can be removed for releasing resources consumed by them. Being in the context of IoT and fog, [46] defines a request's delay as the time spent from an IoT object sends this request until the reception of the request's response. If a request's delay exceeds a predefined timeout, this request is considered as violated. For verifying if a request is violated, this approach assumes that each request is attached with its sending time. Based on this assumption, two heuristics are proposed:

- *min-viol*, which aims at minimizing the number of violated requests;
- *min-cost*, which purposes to minimize the infrastructure provider's financial cost.

As long as an application *app* has violated requests, among edge devices with enough resources to host an instance of *app*, *min-viol* selects the device receiving the most requests to *app* to deploy a new instance. On the other hand, when an application is underloaded, this application's least-charged instance is removed. By modeling the cost of resource consumption and the penalty caused by violations, given an application to reconfigure, *min-cost* adds / removes instances if more benefits can be obtained. This approach deals with IoT applications dependent to sensors' / actuators' locations,

and addresses the dynamicity of adding / removing instances of considered applications, which is equivalent to that of applications' arrival / departure. The proposed heuristics help to make placement decisions rapidly and reactively. However, without discussing instances' migration, this approach is not aware of migration cost. Being designed for applications whose requests are attached with their sending times, [46] is not application-agnostic.

[47] places applications in an infrastructure with network links whose latencies change dynamically. Each considered application is tied to services pinned to certain devices (*i.e.,* these services can not be deployed elsewhere). For lowering communication channels' network latencies, an algorithm is proposed to adjust the placement according to network links' latency changes. For each adjustment, this algorithm tries to migrate each component $c$ to the device at the center of components and pinned services that $c$ communicates with. Considering the non negligible cost of migrations, a component can be migrated only if the improvement (in terms of network latency) brought by this migration exceeds a predefined threshold, which helps to avoid too many migrations. However, the threshold can make it impossible to migrate certain components, even if such migrations help to achieve a global optimum[2]. Moreover, it is difficult to determine the threshold without experience, whose proper value is problem-dependent. This approach deals with the dynamicity of links' latency changes, places IoT applications with location dependency, and is application-agnostic. The proposed heuristic algorithm guides the search to a satisfactory result, which accelerates the search to be reactive to dynamic changes. Without considering all possible migrations (*i.e.,* one migration without significantly lowering communication channels' latencies is not considered), migration cost awareness is considered to be partially supported in [47].

[48] deals with the placement of a single application in an infrastructure composed of edge devices. The considered application is tied to its users' mobiles. Because of users' mobility, the placement must be continuously adjusted to fit the optimization objective—optimizing the application's performance (in terms of response time). [48] proposes an objective function composed of two kinds of costs: *local cost* and *migration cost.* A *local cost* indicates a device's fitness for hosting a component $c$. A device with a lower charge (*e.g.,* CPU utilization rate) and lower network latency to $c$'s users (or their mobiles) fits better $c$. A *migration cost* indicates the penalty for a migration's impacts on the application's performance. A migration between devices close to each other (*i.e.,* in terms of network latency) and between low-charged devices gets a lower penalty. Considering that the decision of a component's host impacts not only the current local / migration costs,

[2]It is possible that a migration without network latency improvement releases resources needed by other components and makes these components' migrations (that can highly lower communication channels' latencies) possible.

but also future ones, [48] purposes to make decisions with future costs taken into account. Based on an assumption of a prediction module, which predicts future local / migration costs, [48] selects the device that minimizes the total cost (*i.e.,* the sum of local and migration costs) during several time slots as a component's next host. Under the dynamicity of device mobility, [48] models the migration cost, makes placement decisions reactively for an application dependent to its users' locations. However, based on an assumption that there is only one application to place, this approach is not application-agnostic.

[49] introduces a distributed mechanism for placing IoT applications in the fog, and continuously adjusts applications' placement because of sensors / actuators' mobility. Aiming at minimizing the infrastructure's bandwidth consumption, [49] takes the following two kinds of costs into account:

- *local cost*, which indicates the bandwidth consumption brought by the communication between components, sensors, and actuators;
- *migration cost*, which indicates the bandwidth consumption caused by migrating components.

By modeling these two kinds of costs, for each component, the device that minimizes the *total cost* (the sum of local and migration costs) is selected as this component's next host. Each component's migration target (*i.e.,* the next host) is calculated by its current host, which makes this mechanism decentralized. Thanks to the decentralization, this approach fits well large-scale infrastructures with a huge amount of devices. However, for avoiding conflicts caused by migrating too many components to a same device (*i.e.,* required resources exceed the device's capacity), devices have to communicate with each other for avoiding such conflicts. Same as [48], [49] also takes future costs into account for further improving placement decisions' quality. Based on a module that predicts sensors / actuators' mobility, a component's migration is planed ahead. A component's migration plan indicates when and to which device this component is going to be migrated. Considering the uncertainty of sensors / actuators' mobility, each component is associated with multiple migration plans, which serve as candidate plans to be selected when mobile sensors and actuators' locations get more certain. However, this mechanism further complicates the conflict avoidance: conflicts must be avoided at any time during the planned period. Moreover, updating one component's migration plans can make it necessary to adjust multiple components' migration plans, and further impacts more and more components, which can make the conflict avoidance mechanism time-expensive. Consequently, although this approach refines the placement decision with several mechanisms, it can lose the reactivity because of the time-expensive conflict avoidance. For the other criteria, this approach deals the dynamicity of devices' mobility, places IoT applications with location dependency, and is application-agnostic.

### Comparison and Summary

According to the criteria introduced in Section 6.1, related works are compared and summarized in Table 3.1.

| | Dynamicity Type Coverage | Migration Cost Awareness | Reactivity | Application Agnostic | Location Dependency |
|---|---|---|---|---|---|
| [42] | device mobility | ✗ | ✓ | ✓ | ✓ |
| [43] | app arrival/departure | ✗ | ✓ | ✓ | ✓ |
| [44] | app arrival/departure | – | ✓ | ✓ | ✗ |
| [45] | app arrival/departure | ✓ | ✗ | ✓ | ✗ |
| [46] | app arrival/departure | ✗ | ✓ | ✗ | ✓ |
| [47] | network link latency | – | ✓ | ✓ | ✓ |
| [48] | device mobility | ✓ | ✓ | ✗ | ✓ |
| [49] | device mobility | ✓ | ✗ | ✓ | ✓ |

Table 6.1: Dynamic Placement Related Work Summary.

[44] and [47] consider components' migration only in specific cases (see the previous subsection for more details). Thus, their migration cost awareness is considered to be partially supported and is noted as – in Table 6.1.

As given in Table 3.1, no related work covers all dynamicity types in the fog (*i.e.,* applications' arrival / departure, devices' mobility, and devices' churn[3]), and none of them is aware of migration cost, reactive, application-agnostic, and deals with location dependency simultaneously. Hence, new mechanisms must be developed to address them.

[3]Considering that the fog is highly dynamic, the latency of any link in the fog can change frequently. To avoid over-committing dynamic changes, our work uses the average latency of each link, which can be considered as static.

# 7

# Proposition for Dynamic Placement

Contents



As stated in previous chapters, this work deals with the problem of placing IoT applications in a fog infrastructure composed of cloud DCs, edge servers, and end devices. To solve such a problem, a placement decision needs to be made, which must:

- map each component (*i.e.,* a software element composing an application) onto a fog node (*i.e.,* a device that provides resources to host applications);
- satisfy constraints for ensuring placed applications' proper execution;
- fit the objective of optimizing applications' performance.

Caused by applications' arrival / departure and devices' mobility / churn, the placement problem in the context of fog and IoT is highly dynamic. Consequently, a placement decision needs to be continuously adjusted (*i.e.,* to change certain components' hosts) for always respecting the constraints and for fitting the optimization objective. Such an adjustment / re-optimization problem is referred to as *dynamic placement problem*.

This chapter gives our proposition for solving the dynamic placement problem, and is organized as follows. Section 7.1 formulates the problem. Section 7.2 introduces our approach for keeping the placement close to the optimum. Section 7.3 presents our approach for quickly repairing the placement when dynamic changes violate certain constraints. Section 7.4 discusses how approaches proposed in Section 7.2 (for placement re-optimization) and Section 7.3 (for violated applications' repairing) can be combined. Finally, Section 7.5 summarizes this chapter.

## 7.1 Dynamic Placement Problem Formulation

Same as the initial placement problem (see Section 4.1 for more details), the dynamic placement problem is modeled with:

- the infrastructure composed of fog nodes, appliances, and links;
- applications composed of components, appliances (or sensing / actuating services provided by and tied to appliances), and bindings.

Same as in initial placement problems, the following constraints must be respected by a solution to a dynamic placement problem:

i) each component is placed in its Dedicated Zone (DZ);
ii) each fog node's CPU / RAM / DISK consumption does not exceed the fog node's capacity;
iii) each link's bandwidth consumption does not exceed the link's capacity;
iv) each binding's latency does not exceed the binding's requirement.

In the fog, a placement decision can be made outdated by changes of the infrastructure and applications (because of constraint violation or non-satisfaction regarding the optimization objective). To keep the placement valid and satisfactory,

- upon *devices' mobility*, components placed in (and sensing / actuating services provided by) moved devices change their locations in the network topology, which can increase certain bindings' latencies / certain links' bandwidth consumption. In this case, certain components must be migrated for repairing (regarding constraint violation) / re-optimizing the placement.
- upon *devices' leaving*, components placed in these disappeared devices must be re-deployed.
- upon *applications' arrival*, for placing newly arrived applications, already placed components can be migrated to release resources needed by these new applications.
- upon *devices' joining / applications' departure*, placed components can be migrated so as to better fit the optimization objective.

As discussed above, a dynamic change can make it necessary to migrate certain components. Migrating a component is resource-consuming and has a negative influence on deployed applications' performance. Simultaneously migrating too many components can even cause a network congestion. Thus, a dynamic placement decision should lead to as few component migrations as possible. Considering that applications' performance is also impacted by their hosts, apart from the number of components to migrate, the quality of selected hosts must be taken into account. In order to optimize selected

hosts' quality, the initial placement approach proposed in Chapter 4 tries to minimize Weighted Average Latency[1] (WAL) of considered applications. To deal with dynamic placement problems, we still use WAL to evaluate selected hosts. Hence, ideally, a dynamic placement decision should minimize both WAL and the number of components to migrate denoted as $\text{NB}_{mig}$.

However, simultaneously minimizing WAL and $\text{NB}_{mig}$ can derive to a contradiction: selecting the optimal hosts regarding WAL can lead to a lot of migrations, while making a decision without migrating components can result in a relatively high WAL. To avoid this contradiction, given a dynamic placement problem, we select the solution leading to the lowest $\text{NB}_{mig}$ among solutions (to this placement problem) with satisfactory WAL values. More precisely, given a set of solutions to the problem, among the ones with WAL lower than a threshold $\text{WAL}_{threshold}$, the solution that minimizes $\text{NB}_{mig}$ is selected as the placement decision. In this work, $\text{WAL}_{threshold} = \sigma \times \text{WAL}_{init}$. $\sigma$ is a predefined parameter[2] ($\sigma > 1$). $\text{WAL}_{init}$ is the WAL of the solution obtained by the initial placement approach (*e.g.,* DAFNO-InitCO-DCO(0.3), see Chapter 4 for more details). $\text{WAL}_{init}$ is re-calculated after each dynamic change, so that the decision can be made according to up-to-date $\text{WAL}_{init}$.

The formulation proposed in this section allows making a placement decision given a set of solutions. How to find out such a set of solutions and how to rapidly find high-qualified ones (regarding the optimization objective) are discussed in Section 7.2.

## 7.2 Dynamic Placement Re-optimization

This section introduces our proposition for dynamically re-optimizing applications' placement in the fog. Genetic Algorithm (GA) [50] is a meta-heuristic inspired by the evolutionary process, which can be used to deal with a wide range of optimization problems, including the placement problem. According to the evaluation of initial placement algorithms given in Chapter 5, GA outperforms FirstFit. Thus, we choose GA as the naive dynamic placement approach, which is detailed in Section 7.2.1. To deal with placement problems, GA generates new placements inheriting[3] from known ones. This feature allows us to design a heuristic that helps to accelerate GA's decision-making process, which is detailed in Section 7.2.2. Another dynamic placement algorithm, which extends the initial placement approach proposed in Chapter 4, is given in Section 7.2.3.

[1] WAL refers to the average latency of bindings with each binding's latency weighted by the binding's bandwidth requirement, see Section 4.1 for more details.

[2] $\sigma$ can be assigned based on a test or experience. How to automatically set (or dynamically update) the value of $\sigma$ remains a future work.

[3] The inheritance is in terms of that, when generating a new placement, a component's host can be selected as the fog node that hosts this component in a kown placement.

### 7.2.1 Genetic Algorithm—A Naive Approach

The pseudo code of GA adapted to the placement problem is given in Algorithm 3. Algorithm 3 continuously refines a population composed of a number of placements. Such a number is assigned by the input *popSize*. Other inputs of Algorithm 3 are: the infrastructure model $infra$, the model of applications to place *apps*, and a probability *proba* (detailed in the following).

**Algorithm 3:** Genetic Algorithm for the Placement

```
   Input: infra, apps, popSize, proba
1  population ← ∅;
2  i ← 0;
3  while i < popSize do
4      p ← getRandomPlacement(infra, apps);
5      if isValid(p, infra, apps) then
6          population ← population ∪ p;
7          i ← i + 1;
8  do
9      best ← population.getBestSolution();
10     backupPop ← population;
11     for each {parent1, parent2} ∈ backupPop do
12         {child1, child2} ← crossover(parent1, parent2);
13         for each child ∈ {child1, child2} do
14             mutate(child, proba);
15             population ← population ∪ child;
16             population.removeWorstPlacement();
17 while best != population.getBestSolution();
18 return best;
```

Line 1–7 initializes *population* as a number of solutions to the considered placement problem. Line 3–7 iteratively adds random solutions into *population* until it contains *popSize* solutions (see line 3). In line 4, *getRandomPlacement*() returns a random placement by mapping each component to a random fog node[4]. In line 5, $isValid()$ checks if the random placement $p$ returned by *getRandomPlacement*() respects all constraints. If so, $p$ is added into *population* (see line 6).

Line 8–17 is an iterative evolution process, which makes solutions stored in *population* fit better and better the optimization objective. In each

[4]Different from FirstFit proposed in Section 4.2, *getRandomPlacement*() does not take any constraint into account. Thus, *getRandomPlacement*() returns rapidly and can be called many times in Algorithm 3.

iteration (*i.e.,* line 9–16), line 10[5] backs up *population* with *backupPop*. *backupPop* is used to control the loop of line 11–16 (see line 11). In line 11, solutions stored in *backupPop* are grouped two by two, and each group is selected as a couple of parents (*i.e.,* $\{parent_1, parent_2\}$) used to generate new placements (*i.e.,* their offspring). Line 11–16 contains three phases:

- *crossover*, which generates offspring based on selected parents;
- *mutation*, in which newly generated offspring mutates in a probabilistic manner to escape from local optimums;
- *selection*, which filters placements that worse fit the optimization objective out from *population*.

In the phase *crossover* (line 12), offspring (*i.e.,* $\{child_1, child_2\}$) is generated by swapping a random set of components' hosts in $parent_1$ and $parent_2$ (as in the example of Figure 7.1). The phase *mutation* (line 14) tests to update each component's host to be a random fog node in the component's DZ according to the input *proba*, which indicates the probability of mutating a component's host. In the phase *selection*, after adding a child into *population* (line 15), the worst placement is removed from *population* (line 16). Such a worst placement is selected according to the following principles (based on the dynamic placement problem formulation given in Section 7.1):

- if an invalid placement (*i.e.,* which violates certain constraints) exists in *population*, this placement is removed;
- otherwise, if a solution's WAL value exceeds $\text{WAL}_{threshold}$, the solution with the highest WAL is removed;
- otherwise, if solutions in *population* have different $\text{NB}_{mig}$ values, the solution with the highest $\text{NB}_{mig}$ is removed;
- otherwise, the solution with the highest WAL is removed.

In this way, only solutions fitting better the optimization objective can be left in *population*, which helps to generate better placements in the next iteration. Iteration by iteration, solutions stored in *population* get closer to the optimum.

The evolution of line 8–17 terminates when the best solution found remains the same after an iteration[6] (see line 17). This termination condition is verified based on the variable *best*, which stores the best solution found till the previous iteration (see line 9). The best solution is returned by *getBestSolution*() and is selected according to the following principles:

[5] Line 9 corresponds to the termination of the evolution, which is detailed later.

[6] GA can have other termination conditions, *e.g.,* when the best solution found is not significantly improved (*i.e.,* the improvement obtained in an iteration is lower than a predefined threshold), or when a timeout is exceeded.

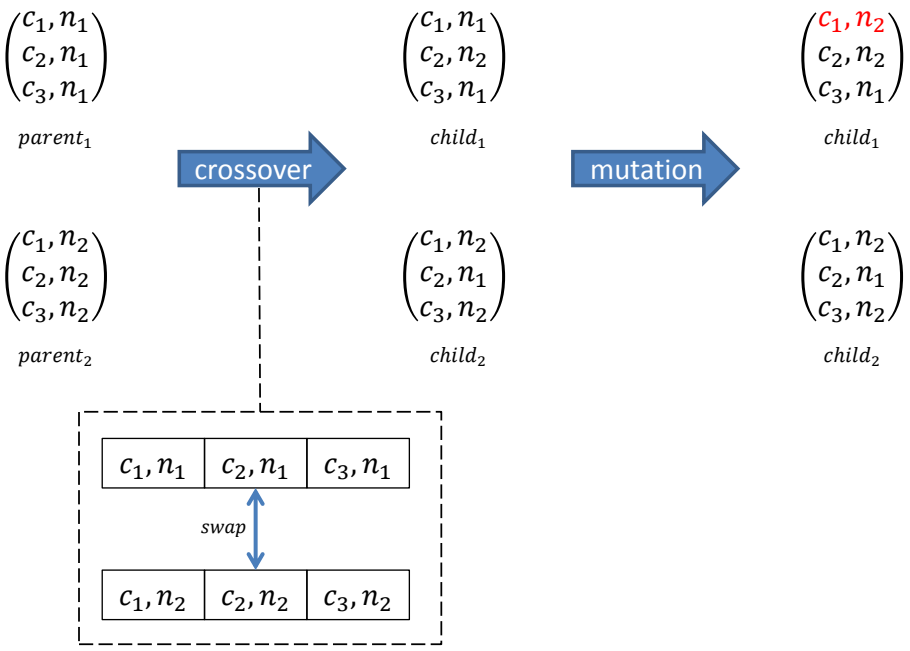


Figure 7.1: Crossover and Mutation Example (based on two placements $parent_1$ and $parent_2$, a host swap is carried out on randomly selected component(s) $c_2$ in the crossover phase. The mutation phase updates components' hosts in a probabilistic manner. In this example, $c_1$'s host mutates to fog node $n_2$).

- if a set of solutions with WAL $<$ $WAL_{threshold}$ is found, among these solutions, the one with minimal $NB_{mig}$ (and minimal WAL[7]) is returned;
- if WAL of each solution in *population* is higher than $WAL_{threshold}$, the solution with the lowest WAL is returned.

Finally, when *best* no longer changes in an iteration, line 18 returns *best* as the placement decision.

Being initialized with random solutions (see line 1–7), GA can need a relatively high execution time to obtain a satisfactory solution, which makes Algorithm 3 impractical for a fog with frequent dynamic changes.

### 7.2.2 Placement Re-optimization Heuristic

As given in Section 7.1, an optimal dynamic placement solution should minimize the number of components to migrate while getting a satisfactory WAL value. Such a solution is similar to two placements: i) the current placement, which does not lead to any component migration; ii) a solution with minimal WAL. Thus, to improve GA, we use these two kinds of placements

[7]WAL is considered when $NB_{mig}$ is minimized by multiple solutions.

as GA's initial placements (instead of random ones in Algorithm 3). The pseudo code of such a heuristic GA is given in Algorithm 4.

**Algorithm 4:** heuristicGA

```
   Input: infra, apps, popSize, proba
 1 p_curr ← getFormattedCurrPlace();
 2 p_init ← initPlace(infra, apps);
 3 population ← {p_curr, p_init};
 4 do
 5 |   best ← population.getBestSolution();
 6 |   backupPop ← population;
 7 |   for each {parent_1, parent_2} ∈ backupPop do
 8 |   |   {child_1, child_2} ← crossover(parent_1, parent_2);
 9 |   |   for each child ∈ {child_1, child_2} do
10 |   |   |   mutate(child, proba);
11 |   |   |   population ← population ∪ child;
12 |   |   |   if population.length > popSize then
13 |   |   |   |   population.removeWorstPlacement();
14 while best != population.getBestSolution() or
         population.size() < popSize;
15 return best;
```

In line 1, *getFormattedCurrPlace*() formats the current placement as follows:

- if a new application arrives, its components' hosts are formatted as empty (which means to be placed);
- if a component's host leaves, this component's host is formatted as empty;
- other components' hosts do not change.

In line 1–3, *population* is initialized to contain the formatted current placement $p_{curr}$ and the solution $p_{init}$ returned by the initial placement approach (*e.g.,* DAFNO-InitCO-DCO(0.3) proposed in Chapter 4). $p_{curr}$ does not migrate any component. $p_{init}$ has a near optimal WAL. By inheriting from $p_{curr}$ and $p_{init}$, offspring generated in Algorithm 4 can rapidly get close to the optimum. Considering that there are only two placements in the initial population, as in line 12, the phase *selection* of GA takes place only when the number of placements in *population* exceeds *popSize*. The loop of line 4–14 terminates when two conditions are simultaneously satisfied (see line 14): i) the best solution found remains the same, and ii) there are at least *popSize* solutions stored in *population*. Other parts of Algorithm 4 are same to Algorithm 3, and are explained in the previous subsection.

Instead of approaching the optimum from random placements, Algorithm 4 uses specific placements to initialize the search. These initial placements are selected with respect to the optimization objective, and can lower the execution time for getting a satisfactory solution.

### 7.2.3 Migration Cost-Aware Heuristics

The initial placement approach proposed in Chapter 4 combines a set of heuristics. Among these heuristics AFNO and DAFNO (see Section 4.3.1 for more details) are responsible for ordering fog nodes. AFNO and DAFNO aims at minimizing WAL values of placement decisions, while components' migration cost is not considered. In order to take WAL and the cost of migrating components into account simultaneously, the following principle is used to extend the initial placement approach: when trying to place a component, its current host (regarding the current placement) must be the first fog node to test. More precisely, after AFNO / DAFNO updates fog nodes' order for placing a component, if this component is currently hosted by a fog node[8], this host is prioritized to be the first fog node to test (*i.e.,* the fog node order produced by AFNO / DAFNO is updated again). Based on this principle, a component will not be migrated if placing this component in its current host can derive to a solution, which helps to migrate as few components as possible. The extended heuristics[9] is compared with Algorithm 3 and Algorithm 4 in Chapter 8.

## 7.3 Reactive Placement Repairing

By violating constraints, dynamic changes in the fog can make deployed applications' execution improper. Such violated applications must be repaired as soon as possible, especially for time-sensitive ones. Algorithms given in Section 7.2 aim at re-optimizing applications' placement, which can be relatively calculation- and time-expensive, and thus may not suit violated applications' repairing. To deal with this problem, this section proposes another mechanism, which aims at making placement decisions as fast as possible.

Among different types of dynamic changes, applications' departure and devices' joining bring more available resources to the infrastructure, which can not make the current placement invalid. When a new application arrives, the status of the fog (*i.e.,* resources provided by the fog and deployed applications' hosts) is not changed, which does not violate the current placement either. Only devices' leaving and devices' mobility can violate deployed

[8]Newly arrived applications are not hosted, and a component placed in a disappeared fog node no longer has a host.

[9]This algorithm extends and is based on DAFNO-InitCO-DCO(0.3), which performs the best among evaluated initial placement approaches according to Chapter 5.

applications, which are respectively taken in charge by two algorithms explained in the following.

When a fog node[10] leaves the infrastructure (*e.g.,* lose network connection, crash out, be turned off), components hosted by this fog node are no longer deployed. Algorithm 5 is responsible for re-placing such components, and is called upon fog nodes' leaving. As Algorithm 5's inputs, $infra$ is the infrastructure's model, and *node* is the disappeared fog node.

**Algorithm 5:** repair4fogNodeLeave

```
   Input: infra, node
 1 nodeList ← infra.fogNodes();
 2 for each comp ∈ node.getHostedComponents() do
 3     sort(nodeList, comp.getAnchor());
 4     isPlaced ← false;
 5     for each n ∈ nodeList do
 6         if not isPlaced and isValid(comp, n) then
 7             place(comp, n);
 8             isPlaced ← true;
 9     if not isPlaced then
10         return "failure";
11 return "pass";
```

In line 1, *nodeList* is assigned as a list containing all fog nodes of the infrastructure (returned by $infra.fogNodes()$). The loop of line 2–10 tries to place each component *comp* hosted by the disappeared fog node *node* (as in line 2). In line 3, fog nodes in *nodeList* are sorted in ascending order of their network latency to *comp*'s anchor[11]. Then, one after another, fog nodes in *nodeList* are tested for hosting *comp* (line 5). When a suitable fog node (verified by $isValid(comp, n)$ in line 6) is found, *comp* is placed in it (line 7). If Algorithm 5 fails to place a component hosted by *node*, "failure" is returned to announce that it fails to repair violated applications (see line 9–10). If Algorithm 5 arrives to place all components hosted by *node*, "pass" is returned (as in line 11). Regarding the order of fog nodes in *nodeList*, Algorithm 5 tries to place each component *comp* in a fog node close to its anchor, which helps to lower WAL. For accelerating the search, Algorithm 5 makes placement decisions only for components hosted by *node* without changing hosts of other components and without guaranteeing to

[10] Because a placer can not successfully place an IoT application with disappeared appliances (or sensing / actuating services), this work focuses on the leaving of fog nodes (*i.e.,* devices that can be used as hosts). The departure of appliances must be addressed by (or with the help of) other modules.

[11] A component's anchor is the barycenter of its network communication. See Section 4.3.1 for more details.

find a solution.

An application *app* is concerned by a device *d*'s mobility if a component of *app* is placed in *d* or one of its sensing / actuating services is provided by *d*. *d*'s mobility can impact latencies and accessible bandwidths of *app*'s bindings, or even violates these bindings' requirements of network resources. Algorithm 6 is responsible for repairing such bindings, and is called upon devices' mobility (that changes the network topology). As Algorithm 6's inputs, $infra$ is the model of the infrastructure, and *device* is the moved device.

**Algorithm 6:** repair4deviceMove

```
   Input: infra, device
1  for each app ∈ device.getConcernedApps() do
2      if isViolated(infra, app) then
3          deplace(infra, app);
4          isFail ← initPlace(infra, app);
5          if isFail then
6              return "failure";
7  return "pass";
```

For each application *app* concerned by *device* (line 1), Algorithm 6 checks whether *app* is violated (line 2). If certain constraints are no longer respected for *app*, by calling *deplace*(), Algorithm 6 deplaces *app* from the infrastructure (line 3). Then, a placement decision for *app* is made by calling the initial placement algorithm (*e.g.,* DAFNO-InitCO-DCO(0.3) proposed in Chapter 4). If $initPlace(infra, app)$ fails to place *app*, "failure" is returned to announce that Algorithm 6 fails to repair violated applications (line 5–6). If all applications violated by *device*'s mobility are repaired, "pass" is returned (as in line 7). Instead of dealing with all applications together (as in initial placement approach proposed in Chapter 4, for the sake of optimizing the result quality and guaranteeing to find an existing solution), Algorithm 6 makes placement decisions for each violated application separately without changing hosts of other ones, which helps to accelerate the placement decision-making process.

Algorithm 5 and Algorithm 6 try to repair violated applications without changing other applications' hosts. Such a mechanism is designed to reduce the problem's search space and to accelerates the search. However, as a cost, these algorithms can only get local optimums, and the placement of considered applications can be worse and worse during a long run. Moreover, these algorithms do not guarantee to find a solution even if it exists. Our approach for solving these problems is given in Section 7.4.

## 7.4 Combination of Re-optimization and Reactive Repairing

Algorithms proposed in Section 7.2 are designed to re-optimize the placement in a dynamic fog, and guarantee to find a solution (if any exists). However, with relatively high complexity, these algorithms may be not reactive enough for repairing violated applications. Differently, Algorithm 5 and Algorithm 6 proposed in Section 7.3 repair violated applications without changing hosts of non-violated ones, which reduce the search space and thus can be highly reactive. Nevertheless, Algorithm 5 and Algorithm 6 have no guarantee on either results' optimality or the ability of finding an existing solution.

To take advantages and avoid drawbacks of the mechanisms given in Section 7.2 and Section 7.3, they can be combined as depicted in Figure 7.2. Algorithm 5 (*resp.,* Algorithm 6) is executed when a device's leaving (*resp.,* mobility) makes the current placement invalid. Algorithms that dynamically re-optimize the placement (*e.g.,* Algorithm 4) can be called when the dynamic change does not violate any constraint and when Algorithm 5 / Algorithm 6 fails to repair the placement.

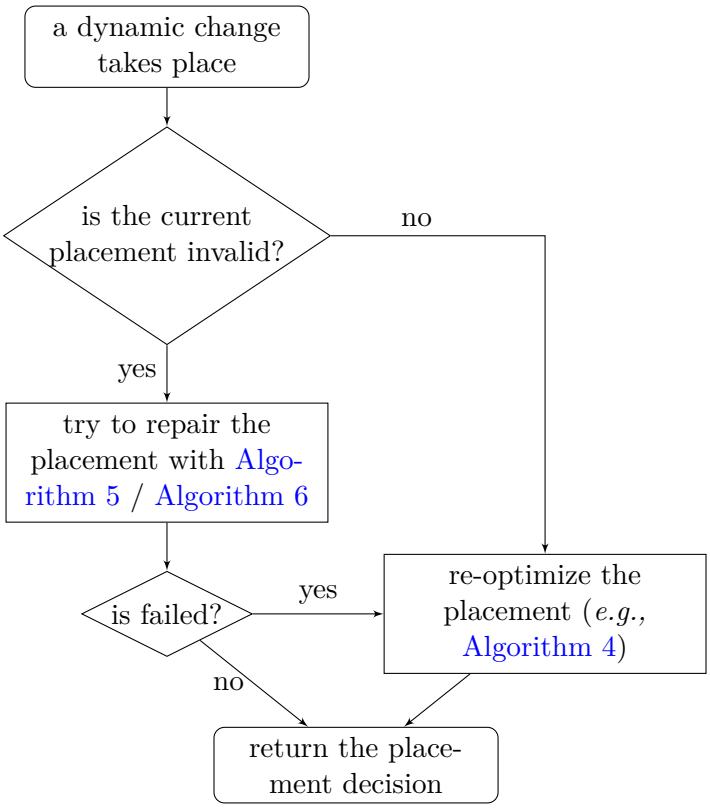


Figure 7.2: Combination of Placement Re-optimization and Repairing Approaches.

In the highly dynamic fog, new dynamic changes can take place during a placement decision-making process (*i.e.,* before a launched algorithm returns). If such a dynamic change violates certain constraints in the newly calculated placement, the new placement must be repaired (by Algorithm 5 and Algorithm 6) before being applied.

The combination of dynamic placements re-optimization and repairing allows them to mutually benefit from each other. Based on the re-optimization, the placement decision can be kept close to the optimum, and an existing solution is guaranteed to be found out. Based on the reactive repairing, violated applications get a chance to be repaired as soon as possible.

## 7.5 Summary

This chapter proposes:

- the dynamic placement problem's formulation;
- a dynamic placement re-optimization mechanism based on GA, and an accompanied heuristic;
- a dynamic placement re-optimization algorithm that extends heuristics proposed to deal with initial placement problems;
- a reactive placement repairing mechanism for quickly adjusting the placement of applications violated by dynamic changes;
- the combination of the re-optimization and reactive repairing mechanisms.

The approach proposed in this chapter supports different dynamicity types (*i.e.,* applications' arrival / departure, devices' joining / leaving, and devices' mobility), is application-agnostic, and helps to make placement decisions close to the optimum and in reactive to dynamic changes violating deployed applications. Algorithms proposed in this chapter are evaluated in Chapter 8.

# 8 Evaluation of Proposed Dynamic Placement Approach

Contents



The problem discussed in this work, how to place IoT applications in the fog, is highly dynamic because of applications' arrival / departure and devices' mobility / churn. In order to cope with such dynamicity, a set of dynamic placement algorithms are proposed in Chapter 7. This chapter evaluates these algorithms based on the use case of Data Stream Processing (DSP) introduced in Section 5.3. The evaluation environment is same as the one used for evaluating initial placement algorithms (see Table 5.1).

In the following, Section 8.1, Section 8.2, and Section 8.3 respectively compare proposed dynamic placement algorithms under: applications' arrival, devices' mobility, and fog nodes' churn. Section 8.4 gives conclusion of this chapter.

## 8.1 Evaluation with Applications' Arrival

In this section, Section 8.1.1 introduces several algorithms that can deal with applications' arrival. Section 8.1.2 evaluates these algorithms with small-scale problems, which allow obtaining and comparing all the algorithms' results. Section 8.1.3 further evaluates these algorithms with large-scale problems.

### 8.1.1 Algorithms to Compare

This section compares: i) two naive algorithms, which respectively consider the cost of migrating components as 0 and $\infty$, and ii) three algorithms

proposed in Section 7.2 for re-optimizing the placement while taking the migration cost into account.

The two naive algorithms are:

- *MigCostZero*. When new applications arrive, MigCostZero makes placement decisions for all components without taking the current placement into account. *i.e.,* MigCostZero is equivalent to the initial placement approach proposed in Chapter 4[1].
- *MigCostInf*. When new applications arrive, MigCostInf makes placement decisions only for components not placed yet (without changing placed components' hosts). *i.e.,* once a component is placed, its host will never be changed[2].

The three algorithms designed for dynamically re-optimizing the placement are:

- *GA* (*i.e.,* Algorithm 3). Based on Genetic Algorithm, GA can find a set of solutions to a dynamic placement problem and returns the one that best fits the optimization objective (see Section 7.1 for more details).
- *HGA* (*i.e.,* Algorithm 4). By extending GA with a heuristic (*i.e.,* initializing GA's population as the current placement and MigCostZero's result), HGA is expected to accelerate the decision-making process of GA.
- *MigCostAware*, which extends the initial placement approach to take the current placement and the cost of migrating components into account (see Section 7.2.3 for more details).

To compare these algorithms, each of them is evaluated from two aspects: scalability and result quality. The scalability is assessed through comparing execution times of different algorithms given the same placement problem to deal with. The evaluation of result quality is based on the comparison of Weighted Average Latency (WAL, see Section 4.1 for more details) and number of components to migrate (*i.e.,* $\text{NB}_{mig}$) obtained by different algorithms.

### 8.1.2 Small-Scale Problems

#### Evaluation Setup

This evaluation reuses the small-scale infrastructure (with 27 fog nodes and 40 appliances) given in Section 5.3.3 (*i.e.,* an infrastructure used in the evaluation of initial placement algorithms based on the DSP use case). Initially,

[1] MigCostZero is based on DAFNO-InitCO-DCO(0.3), which performs the best among evaluated initial placement approaches according to Chapter 5.

[2] MigCostInf's placement decisions made for newly arrived applications are also based on DAFNO-InitCO-DCO(0.3).

one DSP application is placed in this infrastructure. Then, 10 random DSP applications arrive one by one, which implies 10 dynamic placement problems. Each evaluated algorithm needs to make 10 placement decisions. For each evaluated algorithm, after it makes a placement decision, this decision is considered as the current placement when the next application arrives.

### Results and Discussion

Evaluated algorithms' results are given in Figure 8.1. GA's result is not shown because its execution time explodes at the very beginning (*i.e.,* when placing 2 applications with 30 components). Because of the principle that a placed component can not be migrated, MigCostInf does not arrive to find solutions to all the 10 problems. In Figure 8.1 (b), WAL values are normalized according to results of MigCostZero, *i.e.,* given a dynamic placement problem, WAL obtained by MigCostZero is regarded as 1.

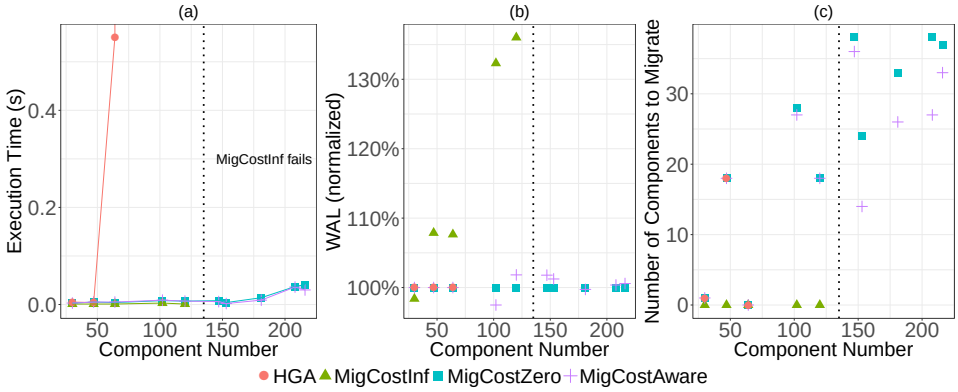


Figure 8.1: Evaluation Results under Applications' Arrival (the dotted line indicates when MigCostInf fails to solve the problem).

According to Figure 8.1 (a), HGA arrives to place 4 applications with 64 components in 0.6s. However, HGA's execution time increases exponentially with the number of components, and gets to be higher than 300s when there are more than 5 applications / 102 components. Compared with HGA, algorithms MigCostInf, MigCostZero, and MigCostAware are much more scalable. In particular, because MigCostInf only needs to place newly arrived applications, its execution time can be even lower than MigCostZero and MigCostAware. However, this difference is rather insignificant in this evaluation.

According to Figure 8.1 (b), HGA, MigCostZero, and MigCostAware get WAL values similar to each other (especially when the number of components is lower than 100). Differently, WAL values obtained by MigCostInf can be much higher. Without considering the possibility of migrating components, MigCostInf can only find local optimums, and can fail to solve certain problems even if solutions exist. In this evaluation, MigCostInf fails

to find a solution when dealing with more than 6 applications. By testing components' current hosts priorly, MigCostAware can miss fog nodes that help to further lower WAL, and thus increases WAL. However, average WAL obtained by MigCostAware is only 0.2% higher than that of MigCostZero, which is rather insignificant.

As shown in Figure 8.1 (c), without migrating any component, MigCostInf's $NB_{mig}$ is always 0. Compared with MigCostZero, MigCostAware helps to get lower $NB_{mig}$ values. The average number of components to migrate obtained by MigCostAware is 13% lower than that of MigCostZero in this evaluation.

### 8.1.3 Large-Scale Problems

#### Evaluation Setup

This evaluation reuses the large-scale infrastructure (with 10561 fog nodes and 20000 appliances) given in Section 5.3.3 (*i.e.,* an infrastructure used in the evaluation of initial placement algorithms based on the DSP use case). Initially, $initAppNB$ randomly generated DSP applications are placed in the infrastructure. Then, applications arrive in 10 rounds, which implies 10 dynamic placement problems. Each evaluated algorithm needs to make 10 placement decisions. For each evaluated algorithm, after it makes a placement decision, this decision is considered as the current placement when the next group of applications arrives. In each round, $appStep$ random DSP applications arrive to be placed. Different combinations of $initAppNB$ and $appStep$ values are discussed in this evaluation ($initAppNB \in \{1, 100, 200, 300\}$, $appStep \in \{2, 5, 10\}$).

#### Results and Discussion

Caused by the large infrastructure scale, GA and HGA get an execution time explosion at the very beginning of this evaluation. Other algorithms' execution times, WAL values, and $NB_{mig}$ values are respectively given in Figure 8.2, Figure 8.3, and Figure 8.4.

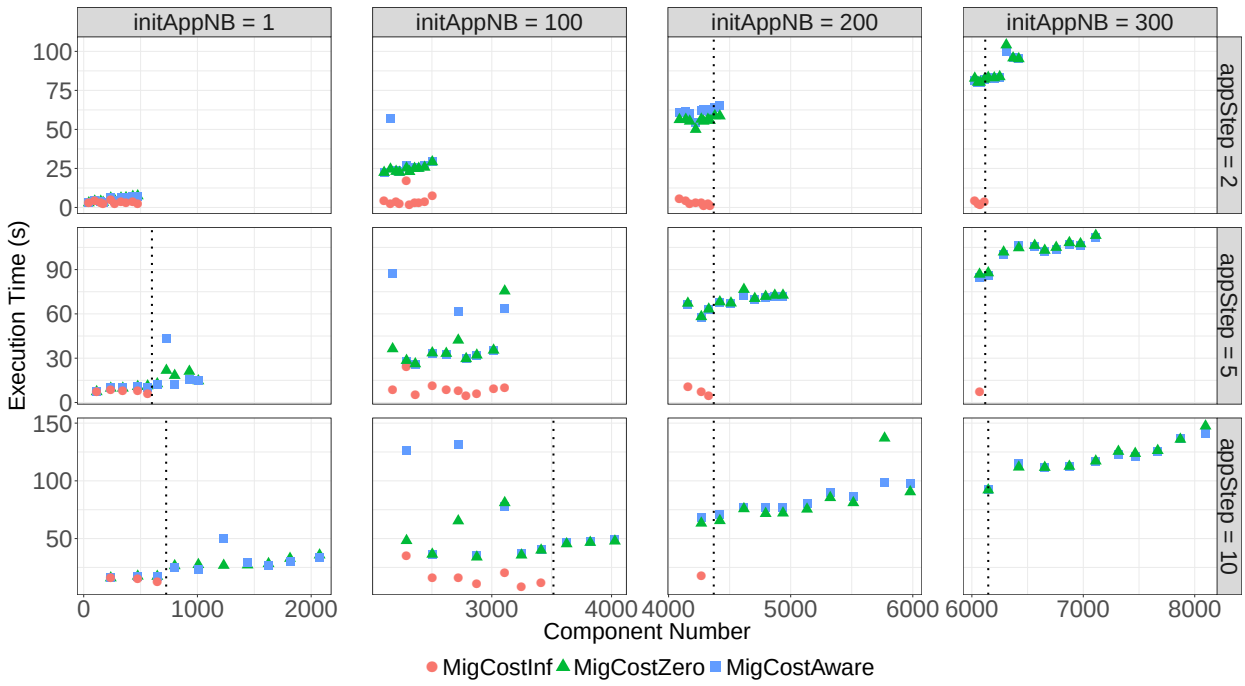


Figure 8.2: Execution Times of Evaluated Algorithms (the dotted lines indicate when MigCostInf fails to solve the problem).

According to Figure 8.2, MigCostZero and MigCostAware get execution times similar to each other, and their execution times increase with the number of components. Differently, MigCostInf's execution time is affected by *appStep* rather than the number of components. Without taking placed components into account, MigCostInf only makes placement decisions for newly arrived applications. Thus, MigCostInf's execution time gets higher when more applications arrive (*i.e., appStep* is higher). Compared with MigCostZero and MigCostAware, MigCostInf's execution time can be much lower especially when there are a lot of components already placed (*e.g.,* MigCostInf gets a 40 times' speed-up when $initAppNB = 300$ and $appStep = 2$). However, without considering the possibility of migrating placed components, MigCostInf can fail to find a solution even if solutions exist, and MigCostInf is more likely to fail when there are more components already placed.

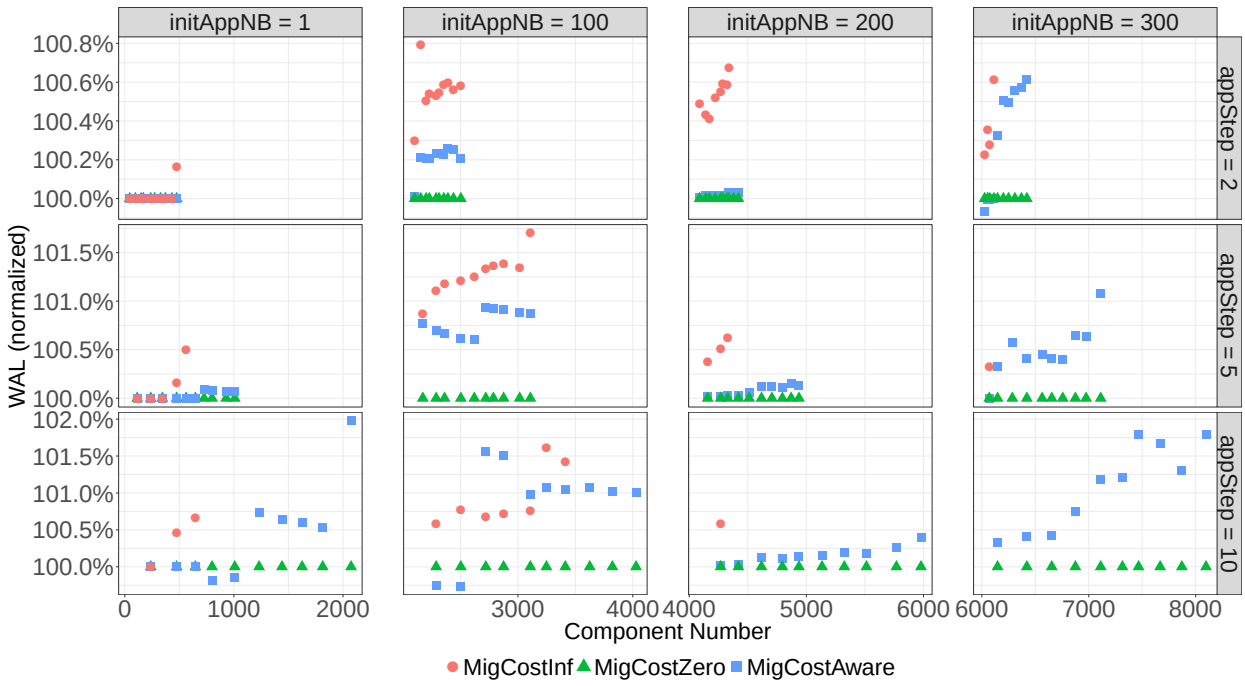


Figure 8.3: WAL Obtained by Evaluated Algorithms.

Evaluated algorithms' WAL values are normalized according to results of MigCostZero, *i.e.,* given a dynamic placement problem, WAL obtained by MigCostZero is regarded as 1. As shown in Figure 8.3, according to problems commonly solved by evaluated algorithms, MigCostInf is prone to get the highest WAL. The difference between WAL values obtained by MigCostInf and MigCostZero increases with the number of arrived applications, which shows that MigCostInf's placement decision can get worse and worse along with applications' arrival. Different from MigCostInf, which solves only a subset of considered problems, MigCostZero and MigCostAware find solutions to all the problems. Among evaluated algorithms, MigCostZero performs the best in terms of lowering WAL values. Nevertheless, the difference between WALs obtained by MigCostAware and MigCostZero is always lower than 2% in this evaluation.

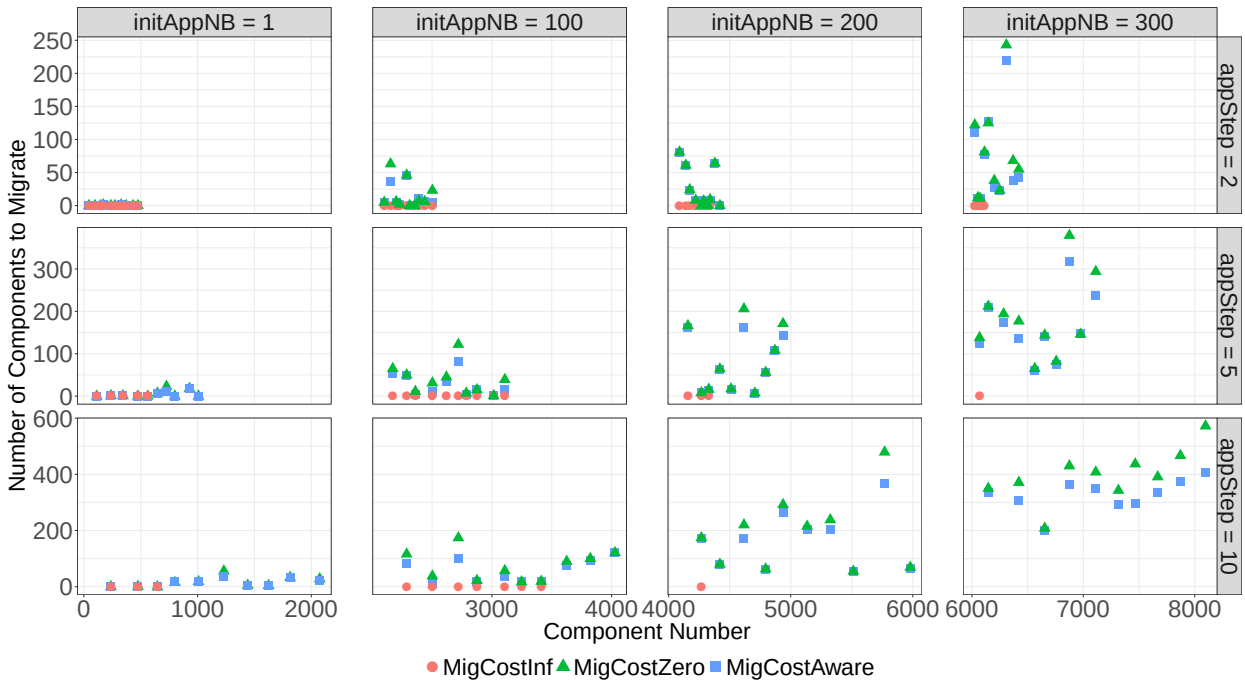


Figure 8.4: MigNB Obtained by Evaluated Algorithms.

MigCostInf does not migrate any component. Thus, its $NB_{mig}$ is always 0. As shown in Figure 8.4, MigCostAware always gets lower $NB_{mig}$ than MigCostZero, and the difference increases with the number of components. When $initAppNB = 300$ and $appStep = 10$, $NB_{mig}$ obtained by MigCostAware is up to 28.8% lower than that of MigCostZero.

The evaluation discussed in this section shows that, when dealing with dynamically arrived applications,

- GA and HGA get satisfactory result quality, however, they are hardly scalable.
- MigCostInf's cost of migrating components is always 0 and has low execution times, nevertheless, it can incur high WAL values and can fail to find solutions to certain problems.
- MigCostZero is much more scalable than GA and HGA. It also highly lowers WAL compared with MigCostInf, nevertheless, MigCostZero leads to a relatively high $NB_{mig}$ values.
- MigCostAware gets execution times and WAL values similar to MigCostZero while highly lowering $NB_{mig}$ values.

## 8.2 Evaluation with Devices' Mobility

This section compares five algorithms[3] GA, HGA, MigCostZero, MigCostAware, and Repair4Mob with the dynamicity of devices' mobility. GA, HGA, MigCostZero, and MigCostAware are explained in Section 8.1.1. When devices move, the current placement can violate certain constraints. Repair4Mob (*i.e.,* Algorithm 6) is designed for rapidly repair (in terms of constraint violation) the current placement upon devices' mobility.

### 8.2.1 Evaluation Setup

This evaluation reuses the large-scale infrastructure (with 10561 fog nodes and 20000 appliances) given in Section 5.3.3 (*i.e.,* an infrastructure used in the evaluation of initial placement algorithms based on the DSP use case). Initially, 400 random DSP applications (with 8097 components) are placed in the infrastructure. Dynamic placement problems are generated by moving[4] a number of end devices (*i.e.,* end fog node / appliance). Different numbers of moved devices (*i.e.,* 50, 100, 200) are discussed in this evaluation. For each of these numbers, 30 dynamic placement problems are generated.

### 8.2.2 Results and Discussion

Because of execution time explosion, we are not able to get results of GA and HGA in this evaluation. Other evaluated algorithms' results are given in Figure 8.5.

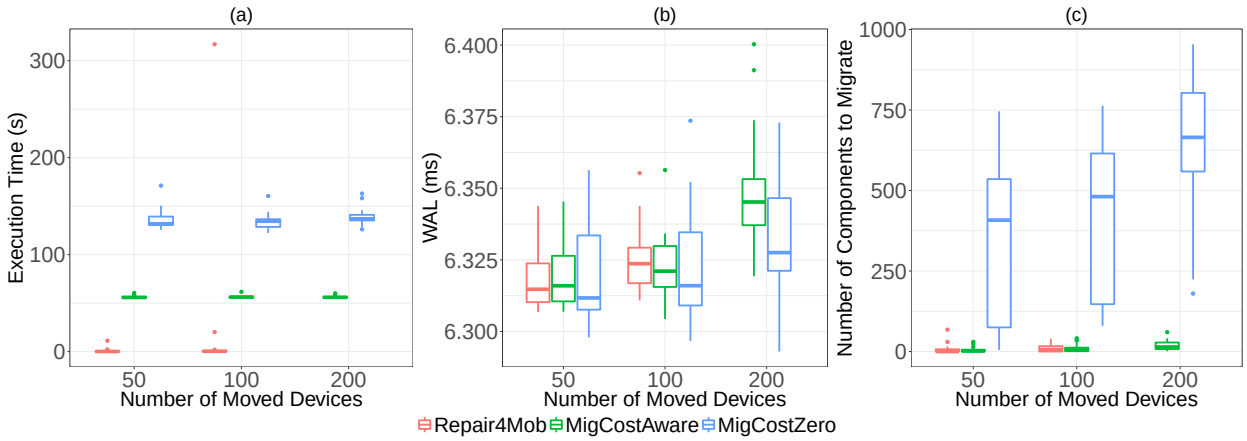


Figure 8.5: Evaluation Results under Devices' Mobility.

[3]MigCostInf is not discussed because it does not update the placement upon devices' mobility, which can result in invalid placements.

[4]After moving an end device, this end device is connected to a random Gateway or Edge Server.

When making a placement decision, MigCostZero and MigCostAware always take all applications into account. Differently, Repair4Mob only updates hosts of applications concerned[5] by moved devices. Thanks to this principle, when there are 50 moved devices, Repair4Mob always gets the lowest execution time among evaluated algorithms (as shown in Figure 8.5 (a)). However, this principle can also make it difficult to find a solution, because resources needed by considered applications (*i.e.,* applications concerned by moved devices) can be taken by other ones. As a result, Repair4Mob's execution time can strongly vary: when the number of moved devices is assigned to 100, although Repair4Mob's execution time is still the lowest when dealing with most of the problems, it gets the highest execution time ($>$ 300s) for solving one problem. Moreover, without considering the possibility of migrating applications not concerned by moved devices, Repair4Mob can fail to solve a problem even if solutions exist (*e.g.,* when the number of moved devices is assigned to 200, Repair4Mob only arrives to find solutions for 5 problems among the generated 30 ones). Both MigCostZero and MigCostAware arrive to solve all generated problems. Devices' mobility can make it necessary to update certain components' hosts. However, for most components, their current hosts are still valid. When trying to place a component, MigCostAware uses the component's current host as the first fog node to test, which helps to start from a valid (in terms of respecting constraints) fog node and thus accelerate the search. As shown in Figure 8.5 (a), MigCostAware gets lower execution times than MigCostZero in this evaluation.

According to Figure 8.5 (b), MigCostZero performs the best in terms of lowering WAL, and the difference of WAL values obtained by MigCostZero and MigCostAware / Repair4Mob increases with the number of moved devices. However, this difference is rather insignificant. When there are 200 moved devices, the average WAL obtained by MigCostAware is only 0.26% higher than that of MigCostZero.

As shown in Figure 8.5 (c), compared with MigCostZero, Repair4Mob and MigCostAware help to highly lower $NB_{mig}$. In this evaluation, MigCostAware gets the lowest $NB_{mig}$, and the average $NB_{mig}$ obtained by MigCostAware is 97% lower than that of MigCostZero.

This evaluation shows that, when dealing with dynamically moving devices,

- GA and HGA are not scalable enough to deal with large-scale problems;
- Repair4Mob highly lowers the execution time when dealing with certain problems, however, it does not guarantee to find existing solutions;
- in this evaluation, MigCostZero arrives to solve large-scale problems in 200s and gets the lowest WAL, however, it leads to a relatively high

[5] An application *app* is concerned by a moved device $d$ if $d$ is an appliance of *app* or *app* has certain components placed in $d$.

$NB_{mig}$;

- MigCostAware gets execution times and WAL values similar to MigCostZero while highly lowering $NB_{mig}$ values compared with MigCostZero.

## 8.3 Evaluation with Fog Nodes' Churn

This section compares five algorithms GA, HGA, MigCostZero, MigCostAware, and Repair4Churn with the dynamicity of devices' churn. GA, HGA, MigCostZero, and MigCostAware are explained in Section 8.1.1. Repair4Churn (*i.e.,* Algorithm 5) is designed for rapidly repair the current placement upon devices' leaving.

### 8.3.1 Evaluation Setup

This evaluation reuses the large-scale infrastructure (with 10561 fog nodes and 20000 appliances) given in Section 5.3.3. Initially, 400 random DSP applications (with 8097 components) are placed in the infrastructure. Dynamic placement problems are generated by making a number of randomly selected end fog nodes leave the infrastructure. Different numbers of disappeared fog nodes (*i.e.,* 25, 50, 100) are discussed in this evaluation, and 30 dynamic placement problems are generated for each of the numbers.

### 8.3.2 Results and Discussion

Because of execution time explosion, we are not able to get results of GA and HGA in this evaluation. MigCostZero and MigCostAware's results are given in Figure 8.6. Repair4Churn's results are listed in Table 8.1 to be discussed specially, because it can only solve a subset of generated problems.

According to the comparison between MigCostZero and MigCostAware shown in Figure 8.6, MigCostZero gets lower WAL values, and MigCostAware gets lower execution times and lower $NB_{mig}$. The average WAL obtained by MigCostAware is only 0.06% higher than that of MigCostZero, while MigCostAware gets an average $NB_{mig}$ that is 46.2% lower than MigCostZero.

Repair4Churn only updates hosts of components placed in fog nodes that leave the infrastructure. This principle helps to accelerate the decision-making process. However, without considering the possibility of migrating components placed in other fog nodes, Repair4Churn can fail to find a solution even if solutions exist. In Table 8.1, *Failure Rate* indicates the number of problems that Repair4Churn fails to solve over 30 (*i.e.,* the number of problems generated for each number of disappeared fog nodes). For each number of disappeared fog nodes, Repair4Churn fails to solve certain problems, and

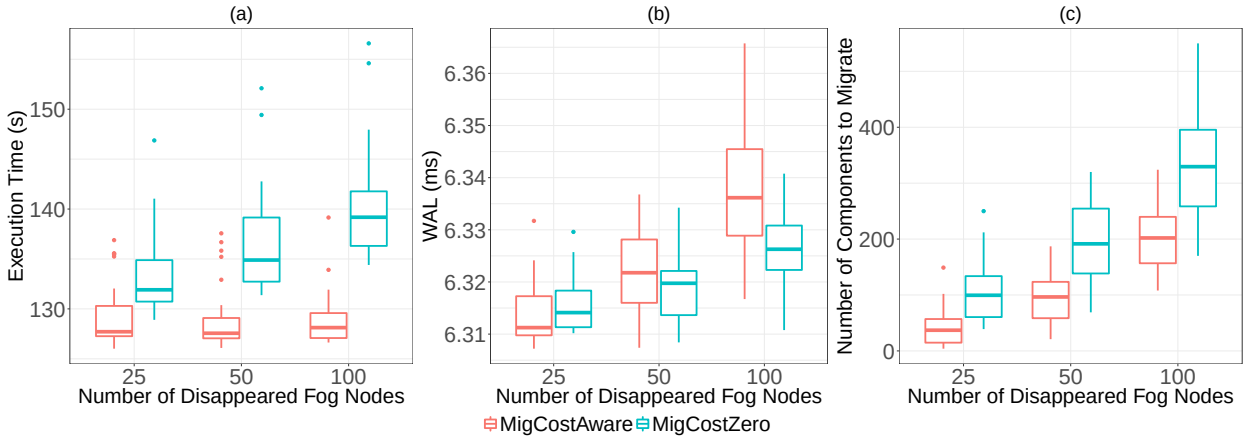


Figure 8.6: Evaluation Results of MigCostZero and MigCostAware under Fog Nodes' Leaving.

the number of problems it fails to solve increases with the number of disappeared fog node. For each problem that Repair4Churn finds a solution, Repair4Churn's execution time and WAL value are normalized according to the result of MigCostZero (*i.e.,* given a dynamic placement problem, the execution time and WAL obtained by MigCostZero are both regarded as 1). Each average of such normalized values is given in Table 8.1 (*i.e., Execution Time (normalized)* for the average of normalized execution times, *WAL (normalized)* for the average of normalized WALs). Compared with MigCostZero, Repair4Churn highly lowers the execution time and gains a speed-up of more than 1000 times. WAL values obtained by Repair4Churn can be higher than that of MigCostZero, and the difference increases with the number of disappeared fog nodes. However, in this evaluation, this difference is rather insignificant. As Repair4Churn does not migrate any component, its $NB_{mig}$ is always 0.

| Number of Disappeared Fog Nodes | Failure Rate | Execution Time (normalized) | WAL (normalized) |
|---|---|---|---|
| 25 | 15 / 30 | 0.054% | 99.99% |
| 50 | 18 / 30 | 0.054% | 100.02% |
| 100 | 27 / 30 | 0.078% | 100.08% |

Table 8.1: Evaluation Results of Repair4Churn under Fog Nodes' Leaving.

This evaluation shows that, when dealing with fog nodes that dynamically leave the infrastructure,

- GA and HGA are not scalable enough to deal with large-scale problems;
- Repair4Churn highly lowers the execution time when dealing with cer-

tain problems, however, it can fail to find solutions to solvable problems (especially when there are a lot of disappeared fog nodes);

- in this evaluation, MigCostZero arrives to solve large-scale problems in 200s and gets the lowest WAL, however, it leads to relatively high $NB_{mig}$ values;
- MigCostAware gets execution times and WAL values similar to MigCostZero while highly lowering $NB_{mig}$ values compared with MigCostZero.

## 8.4 Conclusion

This chapter compares a set of dynamic placement algorithms under different dynamicity types (*i.e.,* applications' arrival, devices' mobility, and fog nodes' churn). According to evaluation results, these algorithms' pros and cons are summarized in Table 8.2.

| **Algorithm** | **Supported Dynamicity Types** | **Scalability** | **WAL** | **$NB_{mig}$** | **Guarantee of Finding a Solution** | **Multiple Solutions' Finding** |
|---|---|---|---|---|---|---|
| GA / HGA | All | ✗ | ✓ | ✓ | ✓ | ✓ |
| MigCostInf | Application Arrival | ✓ | ✗ | ✓ | ✗ | ✗ |
| Repair4Mob | Device Mobility | ✓ | ✗ | ✓ | ✗ | ✗ |
| Repair4Churn | Fog Node Churn | ✓ | ✗ | ✓ | ✗ | ✗ |
| MigCostZero | All | ✓ | ✓ | ✗ | ✓ | ✗ |
| MigCostAware | All | ✓ | ✓ | ✓ | ✓ | ✗ |

Table 8.2: Pros and Cons of Evaluated Algorithms (✓ indicates pros, and ✗ indicates cons).

GA and HGA support all kinds of dynamicity types, and they are able to find multiple solutions, which allows selecting a placement decision according to any objective function (*e.g.,* an objective function with WAL highly weighted or with $NB_{mig}$ highly weighted). Given a placement problem, all the other evaluated algorithms can only find a single definite solution (see *Multiple Solutions' Finding* in Table 8.2). If the found solution is not satisfactory, these algorithms must leverage on other algorithms for making another placement decision. For GA and HGA's disadvantages, their execution times increase exponentially with problem size, which makes them unsuitable to deal with large-scale problems.

MigCostInf, Repair4Mob, and Repair4Churn are respectively designed to deal with one dynamicity type (see *Supported Dynamicity Type* in Table 8.2). Although they can make placement decisions rapidly, it is possible that they fail to find an existing solution (see *Guarantee of Finding a Solution* in Table 8.2). Moreover, compared with other algorithms, MigCostInf,

Repair4Mob, and Repair4Churn are prone to result in high WAL values.

MigCostZero is equivalent to the initial placement approach proposed in Part I, which is able to deal with large-scale problems and to find solutions with low WAL values. However, $NB_{mig}$ is not taken into account in this approach, and thus can be relatively high.

MigCostAware appears as the best compromise taking all the criteria into account. It gets execution times similar to (or even lower than) MigCostZero. Compared with MigCostZero, MigCostAware highly lowers $NB_{mig}$ while introducing an insignificant increase of WAL. The disadvantage of MigCostAware is that, given a placement problem, MigCostAware can find only a single solution. If this solution is not satisfactory (*e.g.,* WAL / $NB_{mig}$ value is too high), it must leverage on other algorithms (*e.g.,* MigCostZero for lower WAL, MigCostInf for lower $NB_{mig}$) for finding out a satisfactory placement. Considering that MigCostInf, Repair4Mob, and Repair4Churn can get execution times even lower than that of MigCostAware in resource-rich infrastructures. If MigCostAware is not reactive enough for certain use cases, it can be combined with MigCostInf, Repair4Mob, and Repair4Churn as stated in Section 7.4.

# 9

# Conclusion and Future Work

Contents



This work tackles the problem of placing distributed IoT applications in the fog, that is how to map a set of software components onto a set of fog nodes. Such a problem is an optimization / search problem with constraints, and is proven to be NP-hard [5, 6]. In order to deal with large-scale problems and to make placement decisions that help to optimize applications' performance, this work focuses on placement algorithms' scalability and placement decisions' quality. The scalability is assessed through problem size (*i.e.,* components' number and fog nodes' number) that an algorithm can deal with under a timeout, and the placement quality is expressed as average response time of considered applications.

## 9.1 Contribution and Discussion

The placement problem is divided into two sub-problems in this work:

- *initial placement problem*, in which only selected hosts' quality needs to be taken into account;
- *dynamic placement problem*, in which both selected hosts' quality and the cost of migrating components must be considered.

As a special case of the dynamic placement problem, an initial placement problem does not have applications already placed in the infrastructure (*i.e.,* the placement decision is made for an infrastructure without any application placed in it). In such a problem, there is no component to migrate and no violated application (*i.e.,* a placed application for which placement constraints are not respected) to repair. For solving the initial placement problem efficiently, it is specially dealt with in this work. The following contributions are made to address the initial placement problem:

- a model and an objective function (*i.e.,* minimizing Weighted Average Latency of considered applications), which formulate the placement problem;
- five heuristics (AFNO, DAFNO, InitCO, DCO, and FailCap) that can be combined with each other;

- a detailed complexity analysis of proposed heuristics;
- a simulation-based evaluation that compares different heuristic combinations and placement algorithms.

The evaluation shows that:

- placements fitting the proposed objective function help to decrease response times of placed applications;
- each proposed heuristic can accelerate the placement decision-making process and / or lower applications' response times;
- the combination of proposed heuristics allows the algorithm to deal with large-scale problems and to make placement decisions close to optimal ones.

The proposed initial placement approach is specifically designed to address IoT applications' placement in the fog. Compared with other placement approaches, this work deals with a wide range of constraints (*i.e.,* CPU, RAM, DISK, devices' properties, bandwidth, and network latency), which ensures that placed applications can be properly executed in a heterogeneous fog infrastructure. By considering that IoT applications are tied to sensors / actuators, the proposition of "anchor" (see heuristics AFNO and DAFNO proposed in Section 4.3.1) allows localizing considered applications (*i.e.,* placing each application in fog nodes close to its sensors / actuators). By taking the existence of resource-constrained devices / links in the fog into account, the heuristic DCO prioritizes components whose resource requirements are hard to satisfy, which helps to accelerate the search. By combining these heuristics, the placement algorithm is highly improved in both scalability and result quality.

To deal with the dynamic placement problem, this work proposes:

- two placement re-optimization approaches, one of which is based on genetic algorithm, and the other one extends the proposed initial placement approach;
- two algorithms, which respectively designed to rapidly repair the placement violated (*i.e.,* certain constraints are not respected) by devices' mobility and churn;
- a mechanism to combine the placement re-optimization and repairing approaches.

The evaluation shows that:

- the proposed dynamic placement approach supports a wide range of dynamicity types (*i.e.,* applications' arrival / departure and devices' mobility and churn), and is able to deal with large-scale problems;

- the placement re-optimization approach allows the placement to be kept close to the optimum (with both selected hosts' quality and the cost of migrating components taken into account);
- the placement repairing approach helps to lower the algorithm's execution time, and thus can make placement decisions rapidly.

## 9.2 Future Work

There are several potential research topics that can be explored based on the work in this thesis:

- *experiment.* To compare different placement decisions, applications' response times are simulated in this work. An experiment on industrial testbeds (such as the Orange Labs internal testbed introduced in [12]) allows measuring application response times in real environments. A further evaluation based on experimentation should be implemented in the future.
- *finer constraints.* This work models an application with constant resource requirements (*e.g.,* a binding's bandwidth requirement is a constant). Considering that resources required by some applications vary periodically (*e.g.,* Smart Bell can require more resources during the day than in the night), the model can be enhanced with time-slotted resource requirements[1].
- *application selection.* When the fog does not have enough resources to host all applications to place, the proposed approach returns "failure" to indicate that these applications can not be satisfied simultaneously. For better dealing with this case, the proposition can be extended with an application selector, which selects a subset of applications to place.
- *automated parameter setting.* In the proposed placement approach, there are several parameters (*e.g.,* $stepLen$, $failNB$) to assign. As a future work, a mechanism that automatically sets these parameters can be developed.
- *multiple optimization objectives.* A placement problem can have multiple optimization objectives. This work only tries to minimize applications' response times. To enhance the proposed approach, other optimization objectives (*e.g.,* minimizing the infrastructure's energy consumption, maximizing placed applications' availability) can be simultaneously considered in the future work.

[1] *i.e.,* dividing a period into multiple time slots, and modeling resource requirements in each time slot.

# Bibliography


[1] Dave Evans. The internet of things: How the next evolution of the internet is changing everything. *CISCO white paper*, 1(2011):1–11, 2011.

[2] Peter Mell, Tim Grance, et al. The nist definition of cloud computing. 2011.

[3] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16. ACM, 2012.

[4] Luis M Vaquero and Luis Rodero-Merino. Finding your way in the fog: Towards a comprehensive definition of fog computing. *ACM SIGCOMM Computer Communication Review*, 44(5):27–32, 2014.

[5] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. Optimal operator placement for distributed stream processing applications. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, pages 69–80. ACM, 2016.

[6] Antonio Brogi, Stefano Forti, and Ahmad Ibrahim. How to best deploy your fog applications, probably. In *International Conference on Edge and Fog Computing*, 2017.

[7] Ye Xia, Xavier Etchevers, Loïc Letondeur, Lebre Adrien, Thierry Coupaye, and Frédéric Desprez. Combining Heuristics to Optimize and Scale the Placement of IoT Applications in the Fog. In *IEEE/ACM International Conference on Utility and Cloud Computing*. IEEE/ACM, 2018.

[8] Ye Xia, Xavier Etchevers, Loïc Letondeur, Thierry Coupaye, and Frédéric Desprez. Combining Hardware Nodes and Software Components Ordering-based Heuristics for Optimizing the Placement of Distributed IoT Applications in the Fog. In *The 33rd ACM/SIGAPP Symposium On Applied Computing*. ACM, 2018.

[9] Neil Gershenfeld, Raffi Krikorian, and Danny Cohen. The internet of things. *Scientific American*, 291(4):76–81, 2004.

[10] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer networks*, 54(15):2787–2805, 2010.

[11] Alessio Botta, Walter De Donato, Valerio Persico, and Antonio Pescapé. Integration of cloud computing and internet of things: a survey. *Future Generation Computer Systems*, 56:684–700, 2016.

[12] Letondeur Loïc, Ottogalli François-Gaël, and Coupaye Thierry. A demo of application lifecycle management for IoT collaborative neighborhood in the fog - Practical experiments and lessons learned around docker. In *Fog World Congress*. IEEE, 2017.

[13] Flavio Bonomi, Rodolfo Milito, Preethi Natarajan, and Jiang Zhu. Fog computing: A platform for internet of things and analytics. In *Big data and internet of things: A roadmap for smart environments*, pages 169–186. Springer, 2014.

[14] Evangelos A Kosmatos, Nikolaos D Tselikas, and Anthony C Boucouvalas. Integrating rfids and smart objects into a unifiedinternet of things architecture. *Advances in Internet of Things*, 1(01):5, 2011.

[15] Jyotiprakash Sahoo, Subasish Mohapatra, and Radha Lath. Virtualization: A survey on concepts, taxonomy and associated security issues. In *Computer and Network Technology (ICCNT), 2010 Second International Conference on*, pages 222–226. IEEE, 2010.

[16] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 273–286. USENIX Association, 2005.

[17] Klervie Toczé and Simin Nadjm-Tehrani. A taxonomy for management and optimization of multiple resources in edge computing. *Wireless Communications and Mobile Computing*, 2018, 2018.

[18] Javad Zare, Saeid Abolfazli, Mohammad Shojafar, and Amirrudin Kamsin. Resource scheduling in mobile cloud computing: Taxonomy and open challenges. In *2015 IEEE International Conference on Data Science and Data Intensive Systems (DSDIS)*, pages 594–603. IEEE, 2015.

[19] Zoltán Ádám Mann. Allocation of virtual machines in cloud data centersa survey of problem models and optimization algorithms. *Acm Computing Surveys (CSUR)*, 48(1):11, 2015.

[20] Mateusz Guzek, Pascal Bouvry, and El-Ghazali Talbi. A survey of evolutionary computation for resource management of processing in cloud computing. *IEEE Computational Intelligence Magazine*, 10(2):53–67, 2015.

[21] Rupali M Pandharpatte. A review: Resource allocation problem in cloud environment. *International Journal of Engineering and Technology*, pages 1695–1700, 2017.

[22] Mohammad Masdari, Sayyid Shahab Nabavi, and Vafa Ahmadi. An overview of virtual machine placement schemes in cloud computing. *Journal of Network and Computer Applications*, 66:106–127, 2016.

[23] Zhiming Zhang, Chan-Ching Hsu, and Morris Chang. Cool cloud: A practical dynamic virtual machine placement framework for energy aware data centers. In *2015 IEEE 8th International Conference on Cloud Computing (CLOUD)*, pages 758–765. IEEE, 2015.

[24] William Tärneberg, Amardeep Mehta, Eddie Wadbro, Johan Tordsson, Johan Eker, Maria Kihl, and Erik Elmroth. Dynamic application placement in the mobile cloud network. *Future Generation Computer Systems*, 70:163–177, 2017.

[25] Olena Skarlat, Matteo Nardelli, Stefan Schulte, and Schahram Dustdar. Towards qos-aware fog service placement. In *Fog and Edge Computing (ICFEC), 2017 IEEE 1st International Conference on*, pages 89–96. IEEE, 2017.

[26] Olena Skarlat, Matteo Nardelli, Stefan Schulte, Michael Borkowski, and Philipp Leitner. Optimized IoT service placement in the fog. *Service Oriented Computing and Applications*, pages 1–17, 2017.

[27] Antonio Brogi and Stefano Forti. Qos-aware deployment of iot applications through the fog. *IEEE Internet of Things Journal*, 2017.

[28] Stamatia Rizou, Frank Diirr, and Kurt Rothermel. Fulfilling end-to-end latency constraints in large-scale streaming environments. In *Performance Computing and Communications Conference (IPCCC), 2011 IEEE 30th International*, pages 1–8. IEEE, 2011.

[29] Cplex toolkit. http://www.ibm.com/support/knowledgecenter/SSSA5P. Accessed: 2018-02-08.

[30] Zhi-Hui Zhan, Xiao-Fang Liu, Yue-Jiao Gong, Jun Zhang, Henry Shu-Hung Chung, and Yun Li. Cloud computing resource scheduling and a survey of its evolutionary approaches. *ACM Computing Surveys (CSUR)*, 47(4):63, 2015.

[31] Mala Kalra and Sarbjeet Singh. A review of metaheuristic scheduling techniques in cloud computing. *Egyptian informatics journal*, 16(3):275–295, 2015.

[32] Ioannis A Moschakis and Helen D Karatza. A meta-heuristic optimization approach to the scheduling of bag-of-tasks applications on heterogeneous clouds with multi-level arrivals and critical jobs. *Simulation Modelling Practice and Theory*, 57:1–25, 2015.

[33] Fatos Xhafa, Javier Carretero, Bernabé Dorronsoro, and Enrique Alba. A tabu search algorithm for scheduling independent jobs in computational grids. *Computing and informatics*, 28(2):237–250, 2012.

[34] Md Hasanul Ferdaus, Manzur Murshed, Rodrigo N Calheiros, and Rajkumar Buyya. Virtual machine consolidation in cloud data centers using aco metaheuristic. In *European Conference on Parallel Processing*, pages 306–317. Springer, 2014.

[35] Jiaxin Li, Dongsheng Li, Jing Zheng, and Yong Quan. Location-aware multi-user resource allocation in distributed clouds. In *Advanced Computer Architecture*, pages 152–162. Springer, 2014.

[36] Sharad Agarwal, John Dunagan, Navendu Jain, Stefan Saroiu, Alec Wolman, and Harbinder Bhogan. Volley: Automated data placement for geo-distributed cloud services. In *NSDI*, volume 10, pages 28–0, 2010.

[37] Xunyun Liu and Rajkumar Buyya. Performance-oriented deployment of streaming applications on cloud. *IEEE Transactions on Big Data*, 2017.

[38] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. Vivaldi: A decentralized network coordinate system. In *ACM SIGCOMM Computer Communication Review*, volume 34, pages 15–26. ACM, 2004.

[39] Maryam Barshan, Hendrik Moens, Steven Latre, Bruno Volckaert, and Filip De Turck. Algorithms for network-aware application component placement for cloud resource allocation. *Journal of Communications and Networks*, 19(5):493–508, 2017.

[40] Henri Casanova, Arnaud Giersch, Arnaud Legrand, Martin Quinson, and Frédéric Suter. Versatile, scalable, and accurate simulation of distributed applications and platforms. *Journal of Parallel and Distributed Computing*, 74(10):2899–2917, 2014.

[41] Marcos Dias de Assuncao, Alexandre da Silva Veith, and Rajkumar Buyya. Resource elasticity for distributed data stream processing: A survey and future directions. *arXiv preprint arXiv:1709.01363*, 2017.

[42] Shiqiang Wang, Kevin Chan, Rahul Urgaonkar, Ting He, and Kin K Leung. Emulation-based study of dynamic service placement in mobile micro-clouds. In *Military Communications Conference, MILCOM*

*2015-2015 IEEE*, pages 1046–1051. IEEE, 2015.

[43] Luiz F Bittencourt, Javier Diaz-Montes, Rajkumar Buyya, Omer F Rana, and Manish Parashar. Mobility-aware application scheduling in fog computing. *IEEE Cloud Computing*, 4(2):26–35, 2017.

[44] Joe Wenjie Jiang, Tian Lan, Sangtae Ha, Minghua Chen, and Mung Chiang. Joint vm placement and routing for data center traffic engineering. In *INFOCOM*, volume 12, pages 2876–2880, 2012.

[45] Rochman Yuval, Levy Hanoch, and Brosh Eli. On dynamic placement of resources in cloud computing. Technical report. Accessed: 2018-02-08.

[46] Ashkan Yousefpour, Ashish Patil, Genya Ishigaki, Inwoong Kim, Xi Wang, Hakki C Cankaya, Qiong Zhang, Weisheng Xie, and Jason P Jue. Qos-aware dynamic fog service provisioning. *arXiv preprint arXiv:1802.00800*, 2018.

[47] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. Distributed qos-aware scheduling in storm. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, pages 344–347. ACM, 2015.

[48] Shiqiang Wang, Rahul Urgaonkar, Ting He, Kevin Chan, Murtaza Zafer, and Kin K Leung. Dynamic service placement for mobile micro-clouds with predicted future costs. *IEEE Transactions on Parallel and Distributed Systems*, 28(4):1002–1016, 2017.

[49] Beate Ottenwälder, Boris Koldehofe, Kurt Rothermel, and Umakishore Ramachandran. Migcep: operator migration for mobility driven distributed complex event processing. In *Proceedings of the 7th ACM international conference on Distributed event-based systems*, pages 183–194. ACM, 2013.

[50] David E Goldberg. Genetic algorithms in search, optimization, and machine learning. *Goldberg.–[USA]: Addison-Wesley*, 1989.